

A Short Cyclopedia of Go

By

John Tullis

Copyright © 2019 John Tullis

All rights reserved.

This book or any portion thereof may not be reproduced, stored in any retrieval system, transmitted in any form or by any means, without the express written permission of the author, except for the use of brief quotations in a book review and certain other noncommercial uses permitted by copyright law.

Table of Contents

A Beginning	7
Abstraction	8
Abstraction: Code Example	10
Algorithm	12
Algorithm: Code Example	12
Channels	14
Channels: Code Example	15
Code Points	18
Code Points: Code Example	19
Composition	20
Composition: Code Example	21
Concurrency	23
Concurrency: Code Example	24
Condition Variable	27
Condition Variable: Code Example	27
Constants	30
Constants: Code Example	30
Data Structures	32
Data Structures: Code Example - Array and Array Pointer	37
Data Structures: Code Example - Linked List	38
Data Structures: Code Example - Struct Field Ordering	39
Data Structures: Code Example - Struct Embedding Struct	40
Deadlock	42
Deadlock: Code Example	43
Declarations	46
Declarations: Code Example	46
Dependency Management	49
Embedding	51
Embedding: Code Example	51
Encapsulation	53
Encapsulation: Code Example	53
Enumeration	55

Enumeration: Code Example	55
Environmental Variables	57
Environmental Variables: Code Example	57
Escape Analysis	59
Escape Analysis: Code Example - Pass By Value	60
First Class Citizen	61
First Class Citizens: Code Example	61
Functions	63
Functions: Code Example	63
Garbage Collection	66
Generics	67
Generics: Code Example	68
Goroutines	71
Goroutines: Code Example	71
Heap	73
Heap: Code Example	73
Inheritance	76
Interface	77
Interface: Code Example	78
Immutability	80
Immutability: Code Example	80
Lexical Scope	82
Lexical Scope: Code Example	83
Linked Lists and Slices	85
Linked Lists and Slices: Code Example	86
Literals	90
Literals: Code Example	91
Logic	93
Logic: Code Example	95
Mapping	97
Mapping: Code Example	98
Marshalling and Unmarshalling	100
Marshalling and Unmarshalling: Code Example - JSON	101

Marshalling and Unmarshalling: Code Example - XML.....	102
Method Set	105
Method Set: Code Example	105
Multiplexing	107
Multiplexing: Code Example.....	107
Mutex (Mutual Exclusion)	110
Mutex: Code Example.....	111
Package	113
Parallelism	115
Patterns	116
Periodicity	119
Periodicity: Code Example	120
Polymorphism	123
Polymorphism: Code Example	123
Race Condition	125
Race Condition: Code Example	125
Recursion and Memoization.	127
Recursion: Code Example	128
Reflection	131
Reflection: Code Example.....	131
Runes	134
Runes: Code Example	134
Stack	136
Templates	137
Templates: Code Example - Text Template	137
Types	139
Types: Code Example.....	140
Type Assertion	142
Type Assertion: Code Example.....	142
UTF-8	144
UTF-8: Code Example	144
Appendix I: Some Comments	146
Appendix II: Additional Code Examples	147

Demonstrate use of error values from user developed functions.....	147
Simple way to load a map with keys and values.....	148
Demonstrates one way to convert a struct to a map.....	149
Demonstrates why closing parentheses are needed for anonymous functions.....	150
Demonstrate a struct with a function field.....	151
Demonstrating the use of maps where the values are structs	152
Demonstrate generation of pseudorandom map keys and values	153
Demonstrate tree creation, loading, traversal.....	154
Sorting a list of planet names not using built in sort function	156
Client Server Two Way Communication.....	158

A Beginning

This book has been created as resource for people interested in the Go language within the context of computer science. The approach taken is to provide computer science and programming concepts, principles, definitions and explanations, and then to relate the language capabilities to each concept or principle. The content is assigned into separate chapters arranged alphabetically, thus this is a cyclopedia.

This book is written to provide support for a range of people, from beginning programmers who are being introduced to concepts of programming and computer science, up to experienced programmers who wish to compare and contrast Go with languages they already know.

This book very useful for people to whom English is a second language, and who need clear explanations of technical vocabulary that is not always well defined from other sources.

In some cases, the Go language uses a rather different approach when providing solutions to a problem domain. Examples include using composition instead of inheritance, ad hoc polymorphism rather than parametric polymorphism, encapsulation via packages and variable capitalization, and the built-in support for concurrency.

Many design patterns exist to compensate for weaknesses in certain programming languages. Where the Go language does not have those weaknesses, Go does not need to implement those patterns. Many Go patterns relate to concurrency.

Some concepts relate to one another in conceptual clusters. For example: concurrency, parallelism, goroutines, and channels form a cluster; as do composition, ad hoc polymorphism, embedding, method sets, and interfaces; while literals, code points, runes, and UTF-8 form another cluster. Reference words are provided within this document to tie concepts with other concepts in a natural cluster. These words map to section titles and are in [blue text](#).

Code examples are deliberately brief. They are provided only to illustrate a presented concept. All code examples have been tested and are compliant with go version 1.11.1. There are a few additional code examples provided in Appendix II. These illustrate interesting programming features of the language that did not warrant a separate topic. All code examples can be executed directly in the Go playground. This is currently located at “play.golang.org” or “play.golang.com”.

To best use the code examples, download Kindle for the PC. You may then copy/paste any and all code examples into the Go playground. You will see they all work properly. Be aware that the playground acts as a single core processor, so while concurrency is available there is no parallelism. Where this impacts the code execution, it is called out in the relevant section.

Abstraction

The word abstraction is derived from the Late Latin word “abstractiōn” which means separation, and this word descends from earlier Latin words “abs” and “trahere” which meant “to draw away from”. In the context of computer science abstraction means filtering out, or hiding, or taking away information that is unnecessary to solve a given problem. Abstraction is applied both to data, and to the control of data.

Data abstraction applies limitation to data - it defines how data is represented and how data is manipulated. From this principle comes the concept of abstract data typing. Every abstract data type specifies data sizing, and an interface which specifies the set of methods or operations used on that type. The term “abstract data type” is normally simplified to “data type”. See [Types](#).

For example, the concept of an “integer” is represented in programming languages as a data type. The data type “integer” is implemented in Go with limitations related to size. Go handles these limitations by making distinctions between signed integers: *int*, *int8*, *int16*, *int32*, and *int64*; and unsigned integers: *uint*, *uint8*, *uint16*, *uint32*, and *uint64*. These are all different data types; *int8* means 8 bits or 1 byte, *int16* means 16 bits or two bytes, and so on. Specifying “*var x int32*” is a declaration of a variable *x* as a data type *int32*.

The reason size is an important component of data typing is that executable programs run on physical machines (even if the program runs in a virtual environment, that virtual environment ultimately runs on physical hardware). This means that all data within a program during runtime must reside in memory or is temporarily resident in physical hardware registers. The compiler must know how to map data entities to the proper sizes so that correct memory allocations are performed when the program is translated down to assembly and then machine code. More information is provided on this topic in the section on [Data Structures](#).

Control abstraction hides the details of the actual processing required to perform operations (actions). The phrase “control instructions” refers to the sequence of instruction statements (flow of control) within a computer that execute a task. A control flow statement is something like “if <something is true> { <do something> } else { <do something else> }”. For example: “*if x > 5 { x = y + 1 } else { x = z }*”.

The details of how the comparison is made, how the determination of whether it is true or false that *x* is greater than 5, and how the assignment operation is performed, is hidden from the programmer. What is really happening under the covers is that compiled source code becomes assembly language which becomes machine code which becomes binary bits (1's and 0's) which are actually electrical charges within the AND and OR gates and data registers of a computer. The details of “how it works” are abstracted (hidden). As is evident, abstraction occurs in multiple levels.

The control abstraction defines the operations that may be performed upon data. To take a simple example, for integers only the following mathematical control operations are permissible: multiplication, division, addition, subtraction, and remainder.

There are levels of abstraction above programming language abstraction, these are generally referred to as “patterns”. A pattern is a general solution for a class of problems, and these are usually classified by level of scope. An architectural pattern applies to computing systems and subsystems, for example: the “Extract, Transform, and Load (ETL)” pattern, or the “Model, View, Controller” pattern. A design pattern (or solution pattern) is applied below the architectural level, and suggests a solution to a problem that will be resolved at the program level. Examples might be Mediator, Pipeline, Producer Consumer, etc. See the section on [Patterns](#).

Patterns that break down big problems into smaller problems often depend upon other patterns and language abstraction. For the Pipeline pattern as an example, in Go this is dependent upon the solution patterns concurrency and parallelism, and the language capabilities of goroutines and channels (control abstraction and data abstraction). See [Concurrency](#) and [Parallelism](#), [Goroutines](#) and [Channels](#). At each layer the abstraction describes what is to occur, but not how it will occur, the how is hidden.

Go was designed to minimize the abstractions built into the language itself. It is quite capable of implementing higher level abstractions, but the language designers chose not to implement certain abstractions into the programming language itself. For example: enumeration, parametric polymorphism, generic types, assertions, exceptions, type inheritance, method overloading, and implicit numeric conversions are not built into the language.

Go is not considered to be a strong language at the level of programmatic abstraction, and this is deliberate. Rather than building abstraction complexity into the language to solve problems, the Go approach has been to create a language with a lower level of control and data abstraction that can effectively support patterns above the language to solve problems.

One capability that Go uses for abstraction all the time is interfaces. There are several topics that relate to interfaces, but the “[Abstraction: Code Example](#)” provides an illustration. The key thing about interfaces is that interfaces are an abstract type because they are described by their behavior (via their methods), and the types that satisfy the interface may implement the methods. Because the methods of an interface contain no code, they are abstract. See [Method Set](#) and [Interface](#).

In the following code example three new data types are defined based on the language types *struct* and *float32*. The new data types are *Square*, *Rectangle*, and *Circle*. For each new data type a single method is declared, each method calculates the area of a variable of the assigned type. Each method has the same name: *Area()*. Finally an interface is declared with one method, also called *Area()*. Therefore each of the three new data types also satisfies the interface, and the interface can be applied to all three data types.

The interface does not reveal how it does what it does, nor does it reveal the underlying representation of the values upon which it operates. This demonstrates interface abstraction. In practice, the usage of this approach is to enclose the data types, the methods, and the interface in a package, and only expose the interface and the data types. When a program imports the package, it may declare variables to be of the specified data types, and call the interface to manipulate the variables, but the importing program will have no insight into how the interface manipulates the variables. This is illustrated more fully in the topics of [Encapsulation](#) and [Dependency Management](#).

See the following simple code for an illustration of abstraction via an interface. This code may be executed in the Go Playground, currently located at play.golang.org.

Abstraction: Code Example

```
// Demonstrate example of interface abstraction
package main

// Package included to permit printing
import "fmt"

// Declare a new datatype
type Square struct {
    side float32
}

// Declare a new datatype
type Rectangle struct {
    length, width float32
}

// Declare a new datatype
type Circle struct {
    radius float32
}

// A method for type Square
func (s Square) Area() float32 {
    return s.side * s.side
}

// A method for type Rectangle
func (r Rectangle) Area() float32 {
    return r.length * r.width
}
```

```
// A method for type Circle
func (c Circle) Area() float32 {
    return 3.14 * (c.radius * c.radius)
}

// Define interface abstractly, hiding how it operates on variables
type Areas interface {
    Area() float32
}

// Demonstrate interface abstraction - identical call, different results
func main() {
    // Declare and assign value or values to variables
    s := Square{side: 4.5}
    r := Rectangle{length: 5.2, width: 3.5}
    c := Circle{radius: 3.25}

    fmt.Println("Square sides are: ", s)
    fmt.Println("Rectangle sides respectively: ", r)
    fmt.Println("Circle radius is: ", c)

    // Define a variable of type interface
    var a Areas

    // Assign to the interface a variable of type Square
    a = s
    // Call the interface
    fmt.Println("Area of Square: ", a.Area())

    // Assign to the interface a variable of type Rectangle
    a = r
    // Identical call to the interface
    fmt.Println("Area of Rectangle: ", a.Area())

    // Assign to the interface a variable of type Circle
    a = c
    // Identical call to the interface
    fmt.Println("Area of Circle: ", a.Area())
}
/*
Prints the following:
Square sides are each: {4.5}
Rectangle sides respectively : {5.2 3.5}
Circle radius is: {3.25}
Area of Square: 20.25
Area of Rectangle: 18.199999
Area of Circle: 33.166252
*/
```

Algorithm

An algorithm may be defined as a set of rules specified in a well-defined formal language that provides a solution to a problem via a finite number of steps. The word is derived from a combination of the Medieval Latin word “*algorismus*” (which originated from Arabic) and the Greek word “*arithmos*” (which means number).

To solve a problem, an approach is to break the problem into smaller pieces (sub-problems). Where any of the smaller pieces is not small enough, functionally decompose it into even smaller pieces. Within computer science, this act of decomposition is also called factoring. This word derives from the Latin “*factor*” which means “to make”.

Once the problem is sufficiently decomposed into separate smaller problems, the solution is built up by applying algorithms (the finite sequence of steps) to each sub-problem. Each subproblem is assigned its own algorithm, and if any problem requires more than one algorithm then it is further decomposed into subproblems until each subproblem only has one algorithm assigned to solve that problem.

Go is a well-defined procedural language used to apply algorithms to solve problems. With a problem decomposed into pieces, individual areas of functionality are assigned to solve each piece of the problem. Each functional area (usually implemented as functions or methods) should be designed to implement a single algorithm in code. Where those Go functions can operate independently of one another, they can become goroutines. See [Functions](#), [Goroutines](#), and [Method Set](#).

Consider a simple problem of data manipulation. This breaks down into input (get the information), process (manipulate data), and output (write out the information somewhere). Taking the input subproblem - is it from a file, a database, or provided via a calling requestor (network based input)? Assume it is a file. Then this breaks down into sub-subproblems: finding the file (where is the file), opening the file (requires access permissions), and reading data from the file (is the data in lines, records, a continuous string of data, or binary, and so on).

So, to limit the scope, assume the subproblem relates to file access. Typical things that must be handled are: file name, access permissions, file creation, file reading and writing, and error handling. For a simple example of file creation, assigning permissions, writing, reading, and error handling operations, see the following code example:

Algorithm: Code Example

```
// Demonstrate the algorithmic concept in the context of file input/output
package main

import (
```

```
"fmt"    // Need for printing
"io/ioutil" // Need to be able to do file read/write operations
"log"     // Need for logging of errors
)

func main() {
    // Specify a filename and path
    f := "/tmp/myfile.txt"

    // Create file, assign permission, and write to it
    err := ioutil.WriteFile(f, []byte("foobar barfoo\n"), 0644)
    if err != nil { // err is == nil when call is successful
        // Fatal is like Print() followed by a call to os.Exit(1)
        log.Fatal(err)
    }

    // Read from file, place contents in variable v
    v, err := ioutil.ReadFile(f)
    if err != nil { // err is == nil when call is successful, which it will be
        log.Fatal(err)
    }

    // Print file name, print file contents, v is type []byte, requires string conversion
    fmt.Println("File name is: ", f)
    fmt.Println("File data is: ", string(v))

    // Specify file that does not exist
    f = "/tmp/nosuchfile.txt"
    v, err = ioutil.ReadFile(f) // Read from file (this will fail!)
    if err != nil { // err will be != nil since no such file to read
        log.Fatal(err) // Since file does not exist, error condition occurs
    }
}

/*
Prints something like the following:
File name is: /tmp/myfile.txt
File data is: foobar barfoo
<date time stamp> open /tmp/nosuchfile.txt: No such file or directory
*/
```

Channels

Channel means the the course through which something is directed. It is derived from the Middle English word “*chanel*” which derived from the Latin “*canālis*” meaning “waterpipe”. In computing pipes permit communication between processes, between threads, or between lightweight threads (a thread may have many lightweight threads). In the context of Go a channel is a pipe that enables communications between concurrent goroutines.

A channel is created via calling “*make(chan type)*”, for example, “*make(chan int)*” or “*make(chan int, 10)*”. A channel may be either unbuffered or buffered. The difference is that unbuffered channels enforce synchronous communication via the channel between two or more goroutines, while buffered channels permit asynchronous communication between goroutines. To create an synchronous channel, do not give it a size; to make it asynchronous, give it a size. The size specifies how many values of the designated type the channel will hold.

When a goroutine sends data to a unbuffered channel, it then blocks until another goroutine does a receive from that channel; however if the goroutine sends data into a buffered channel it does not block unless the channel is full. For example, a channel defined like “*ch1 := make(chan int, 10)*” can hold 10 integer values, if a sender tries to place an 11th value into this channel when it already has 10 values, then the sender will block.

When a goroutine performs a receive on either a synchronous or asynchronous channel, it will block until the channel receives a sent value, unless performing the channel read within a select statement. This reading within a select statement is performed when a goroutine does not want to block when receiving on a channel. Typically this situation occurs where the goroutine has other work to do other than channel processing, or where it might be monitoring multiple channels. For an example of this type of channel operation, see: [Multiplexing](#).

While it is safe to attempt to read from a closed channel, attempting to write to a closed channel will cause a panic condition. One implication is that it is safer for a goroutine that writes to a channel to be responsible for closing the channel, and that there be only one writer to a channel. But if it is necessary for multiple writers to have write access to a channel, then a mutual exclusion lock can be used to control write access to the channel. See [Mutex](#).

Channels may either be declared as global variables, or by the calling function (such as within the main goroutine). The recommended practice is that the channels are passed as arguments to the goroutines which will use them.

Channels may be specified to be bidirectional or unidirectional. Channels may be declared to be unidirectional for a goroutine by specifying this in the format of the channel parameter in the goroutines argument list during the function declaration.

The following code example demonstrates the use of synchronous and asynchronous channels, passing channels as parameters to goroutines that will use the channels, and specifying passed

channels as being read only, write only, or both. The select statement used here in the goroutines shows the use of one case to check an input channel but if nothing is there, to go to a second case and simulate doing some non-channel related work.

The [Goroutines](#) section illustrates another example using channels, also see [Concurrency](#) and [Parallelism](#).

Channels: Code Example

```
// Demonstrate basic functionality of channels
package main

import (
    "fmt"
    "time"
)

// This goroutine receives a read only buffered asynchronous channel
func mygo1(ch <-chan string) {
    fmt.Println("mygo1 started ")
    for {
        select {
        case s := <-ch:
            fmt.Printf("mygo1 received message %s\n", s)
            // Simulate work after receiving channel notification
            time.Sleep(1000 * time.Millisecond)
        case <-time.After(2000 * time.Millisecond):
            fmt.Println("mygo1 timed out waiting for message!")
            // Would do other non channel related work here
        }
    }
}

// This goroutine receives a read and write buffered asynchronous channel
func mygo2(ch chan string) {
    fmt.Println("mygo2 started ")
    for {
        select {
        case s := <-ch:
            fmt.Printf("mygo2 received message %s\n", s)
            // Simulate work after receiving channel notification
            time.Sleep(1000 * time.Millisecond)
        case <-time.After(1000 * time.Millisecond):
            fmt.Println("mygo2 timed out waiting for message!")
            // Would do other non channel related work here
        }
    }
}
```

// ch1 is a read/write buffered asynchronous channel, ch2 is a write only unbuffered synchronous channel

```
func mygo3(ch1 chan string, ch2 chan<- string) {
    fmt.Println("mygo3 started ")
    for {
        select {
        case s := <-ch1:
            fmt.Printf("mygo3 received message %s\n", s)
            // Simulate work after receiving channel notification
            ch2 <- "mygo3 exits after processing only one message"
            return // Exit goroutine here
        case <-time.After(3000 * time.Millisecond):
            fmt.Println("mygo3 timed out waiting for message!")
            // Would do other non channel related work here
        }
    }
}
```

```
func main() {
    ch1 := make(chan string, 3) // Create buffered asynchronous channel
    ch2 := make(chan string)    // Create unbuffered synchronous channel
    go mygo1(ch1)
    go mygo2(ch1)
    go mygo3(ch1, ch2)
```

```
    // Give goroutines time to start
    time.Sleep(1000 * time.Millisecond)
    ch1 <- "Message One"
    ch1 <- "Message Two"
    ch1 <- "Message Three"
```

```
    // Wait a bit for goroutines to run
    time.Sleep(2000 * time.Millisecond)
    ch1 <- "Message Four"
    ch1 <- "Message Five"
    ch1 <- "Message Six"
    s := <-ch2 // Read response from mygo3
    fmt.Println(s)
```

```
    // Pause to show goroutines waiting for messages
    time.Sleep(6000 * time.Millisecond)
    fmt.Println("Exiting now")
}
```

/*

Prints something like:

```
mygo1 started
mygo2 started
mygo3 started
mygo2 timed out waiting for message!
```



```
mygo2 received message Message Three
mygo1 received message Message One
mygo3 received message Message Two
mygo2 timed out waiting for message!
mygo3 exits after processing only one message
mygo2 received message Message Five
mygo1 received message Message Four
mygo2 received message Message Six
mygo2 timed out waiting for message!
mygo1 timed out waiting for message!
mygo2 timed out waiting for message!
mygo2 timed out waiting for message!
mygo1 timed out waiting for message!
mygo2 timed out waiting for message!
Exiting now
*/
```

Code Points

Code points are the numerical values that make up a code space for character encoding, where the code space contains one or more character sets. Within the context of the Go language, the code space is specified as the UTF-8 character coding set. See [UTF-8](#).

Each code point is considered to reference a single “character”. The UTF-8 code points may be between one and four 8-bit bytes (called octets). The entire ASCII character set is represented by 128 single byte code points. The characters of all known languages, all mathematical symbols, and all ideograms, can be represented by the code points within the UTF-8 code space.

Go source code is composed of characters that map to UTF-8. This means that the Go source code may contain any known character and may represent new characters as they are invented and assigned into the UTF-8 code space.

Within Go documentation, the UTF-8 code points are often referred to as “runes”. The term rune and the term UTF-8 code point are synonymous in meaning (except when *rune* means data type, see below).

This matters because a string of characters, which is a sequence of bytes, may contain non-ASCII characters. There is not a one to one relationship between a byte and a character within a string. A given string character may exist in the code within one byte, or two bytes, or three bytes, or even four bytes. Therefore, a UTF-8 code point varies in size.

While runes and UTF-8 code points mean the same thing, the Go language specification *rune* type is *int32*, which is four bytes. So, while a “rune” refers to a symbol representation which may vary from one to four bytes in the UTF-8 encoding space, the *rune* alias within the Go language refers to a type of *int32*. The purpose of the rune type within the Go language is so that any UTF-8 code point may be stored, without knowing in advance the actual size of the code point. Because UTF-8 code points may be at most four bytes, the *int32* type is used for this purpose.

Glyphs are composed of runes and so may be indirectly represented via the code space because the code points of the characters which make up the glyph are in the code space. The term glyph comes from Greek and is derived from *glýphein* which means “to hollow out” or “to engrave”.

Go has a package called “strings”, which implements functions to manipulate UTF-8 encoded strings. For example, the strings package function *ContainsRune()* determines if a single given Unicode code point (rune) exists in a string. The strings package function *ContainsAny()* determines if any Unicode code points in one string also exist in a second string, either as a sequence or as individual characters.

The following example code shows how ASCII and non-ASCII characters are represented within string literals, including an ideographic symbol. The “*strings*” package is imported to show how code point information can be inspected. See [Runes](#) for information regarding how runes relate to characters and glyphs. Note: while Go handles all of the UTF-8 characters, the ebook format does not, so the following example uses an ideogram that the ebook can handle: the 🍌.

Code Points: Code Example

```
// Show code points containing characters and ideograms
package main

import (
    "fmt"
    "strings"
)

func main() {
    // ContainsRune finds whether a string contains a particular Unicode code point
    // Specification: func ContainsRune(s string, r rune) bool
    fmt.Println("Is 'v' inside 'aardvark':", strings.ContainsRune("aardvark", 'v'))
    fmt.Println("Is 'v' inside 'buffalo':", strings.ContainsRune("buffalo", 'v'))
    fmt.Println("Is '🍌' inside 'a 🍌 b':", strings.ContainsRune("a 🍌 b", '🍌'))

    // ContainsAny reports whether any Unicode code points in string chars are within string s
    // Specification: func ContainsAny(s, chars string) bool
    fmt.Println("Is string '🍌' inside 'a 🍌 b':", strings.ContainsAny("a 🍌 b", "🍌"))
    fmt.Println("Are both 'b & r' inside 'foobar':", strings.ContainsAny("foobar", "b & r"))
    fmt.Println("Is 'oba' inside 'foobar':", strings.ContainsAny("foobar", "oba"))
    fmt.Println("Is string '🍌' inside 'foobar':", strings.ContainsAny("foobar", "🍌"))
    fmt.Println("Is '\"' inside 'foobar':", strings.ContainsAny("foobar", ""))
    fmt.Println("Is '\\\"' inside '\\\"':", strings.ContainsAny("", ""))
}
/*
```

The above code produces the following results:

```
Is 'v' inside "aardvark": true
Is 'v' inside "buffalo": false
Is '🍌' inside "a 🍌 b": true
Is string "🍌" inside "a 🍌 b": true
Are both "b & r" inside "foobar": true
Is "oba" inside "foobar": true
Is string "🍌" inside "foobar": false
Is "" inside "foobar": false
Is "" inside "": false
*/
```

Composition

Composition means the act of combining different elements into a whole, and the term is originally from Latin (*compositiō*) via Middle English (*composicioun*). In computer science composition is the principle that entities can achieve polymorphism by incorporating other entities that contain the desired functionality. This approach focuses on behavior (what an entity can do) rather than content (what an entity has or owns). See [Polymorphism](#).

Since the Go design approach to composition is focused on what a type can do, rather than on what a type is, it avoids the hierarchical approach of inheritance. Go does not implement either classes or inheritance, but by combining types with interfaces, Go enables similar functionality without the overhead or the complexity.

Go demonstrates different forms of composition: functional composition and type composition, where type composition includes both data type composition and interface type composition

Type composition is implemented within the language via the technique of embedding. Composition and embedding are not the same thing. Composition is a design principle. Embedding is a programmatic language technique implementing the principle.

Data type composition is implemented when a data type embeds another data type. That other data type may or may not have embedded interface methods. The way this is done in Go is a two step process.

1. Create within a data type (often a struct) what is called an embedded or anonymous field, where this is a field that has a type but no name, and the data type has been previously declared
2. If the data type that is embedded has implemented an interface, then the data type that has embedded another data type can also access the interface method or methods, and these methods are referred to as “promoted” within the context of the embedding type.

Rather than include the data type composition code example here, instead see [Embedding](#).

Interface composition is the second aspect of type composition. This is also type composition because interfaces are types, they are types that contains zero or more methods. Assuming the interface has methods, a data type (often a struct but not always) may do a method declaration on an interface to access the methods, this is called embedding an interface. A type is considered to “satisfy” an interface if it incorporates all the methods of an interface. Also, one interface may embed one or more other interfaces.

The interface composition code example is provided in the interface topic, see [Interface](#).

Functional composition means that results from function calls can be returned to other functions. For example, function *a()* may be passed as a parameter to function *b()*, as *b(a())*.

This may be extended to $c(b(a()))$, and so on. Go supports this form of composition. Further examination of function composition is shown as follows:

Composition: Code Example

```
// Demonstrate function composition
package main

import (
    "fmt"
)

func a(x, y int) (int, int) {
    fmt.Printf("Function a() received %d %d.\n", x, y)
    x = x + x
    y = y + y
    fmt.Printf("Function a() returning %d %d.\n", x, y)
    return x, y
}

func b(x, y int) (int, int) {
    fmt.Printf("Function b() received %d %d.\n", x, y)
    x = x + y
    y = x * y
    fmt.Printf("Function b() returning %d %d.\n", x, y)
    return x, y
}

func c(x, y int) (int, int) {
    fmt.Printf("Function c() received %d %d.\n", x, y)
    x = y / 2
    y = y * x
    fmt.Printf("Function c() returning %d %d.\n", x, y)
    return x, y
}

// Demonstrating functional composition
func main() {
    x, y := c(b(a(1, 2)))
    fmt.Printf("Received: %d %d.\n", x, y)
}

/*
Prints the following:
Function a() received 1 2.
Function a() returning 2 4.
Function b() received 2 4.
Function b() returning 6 24
Function c() received 6 24.
```

Function c() returning 12 288.
Received: 12 288
*/

Concurrency

Concurrency comes from the Medieval Latin word “*concurrentia*” meaning “to run together”, and in the context of computer science it means that a program may call multiple functions and not wait for any of those functions to return. In Go this kind of function is called a goroutine. Rather than sequential logic (flow of control) where a series of sequential function calls are all pushed onto the same stack; with concurrent logic multiple stacks are in operation. See [Stack](#).

Concurrency does not mean parallelism. Programs with concurrency can run on a single processing core, while parallelism requires multiple cores. See [Parallelism](#).

If 100% of program execution is done in the processor (no input/output), then concurrency may be unnecessary. The concurrent program will run just slightly slower than in purely sequential logic due to the context switching required to load the separate stacks into and out of the processor core. For example: Assume there is a main goroutine that takes 1 second, a myf1 goroutine that takes 10 seconds of processor time, and a myf2 goroutine that takes 10 seconds of processing time. Total time is $1 + 10 + 10 = 21$ seconds, plus some context switching.

However, if program execution is not 100% processor bound, then performance improvements are possible even if the program only has access to a single processor core. Assume there is a main goroutine that takes 1 second, a myf1 goroutine that interleaves 5 seconds of processing time with 5 seconds of I/O time, and a myf2 goroutine that interleaves 5 seconds of processing time with 5 seconds of I/O time. Assume the interleaving segments are each 1 second. Then the total program processing time can be reduced to 12 seconds (1 second for main, and 11 seconds for myf1 and myf2), plus some context switching time. See the following image:

	1 sec	1 sec	1 sec	1 sec	1 sec	1 sec	1 sec	1 sec	1 sec	1 sec	1 sec
myf1	Core	I/O	Core	I/O	Core	I/O	Core	I/O	Core	I/O	
myf2		Core	I/O	Core	I/O	Core	I/O	Core	I/O	Core	I/O

Goroutine time sequence (excluding context switching overhead) = 11 seconds

Program performance is not the only reason to use concurrency. If a program has several distinct actions for which it is responsible, encapsulating each separate responsibility into a separate concurrent area of functionality implements the Single Responsibility Principle. So for example when communicating with a database:

1. One area of responsibility is handling the database interface (connection, connection pooling, query language communication, disconnection)
2. Another area of responsibility would be formatting of data sent to and received from the database into and out of internal data structures
3. A third area of responsibility is accepting requests from entities needing to communicate with the database, and responding to those entities with data from the database

Go permits you to handle these separately by encapsulating each separate responsibility into goroutines. See [Goroutines](#).

The following code example implements a simple client and server scenario. The main goroutine launches a server goroutine and two client goroutines. Whenever the server receives a connection request from a client, it launches a connection handler and passes the connection established with the client to the connection handler, which then handles communication.

Concurrency: Code Example

// Demonstrates concurrency with a server handling multiple clients

package main

import (

"fmt"

"io"

"log"

"net"

"os"

"time"

)

// Client goroutine

func simulateClient1() {

// Connect to a server

conn, err := net.Dial("tcp", "127.0.0.1:8000")

if err != nil {

log.Fatal(err)

}

// Ensure server disconnect when done

defer conn.Close()

// Communicate with the server

for i := 1; i < 6; i++ { // Loop 5 times

conn.Write([]byte(fmt.Sprintf("Client 1 sent message #%d\n", i)))

time.Sleep(1000 * time.Millisecond)

}

conn.Write([]byte("Client 1 exiting\n"))

}

// Client goroutine

func simulateClient2() {

// Connect to a server

conn, err := net.Dial("tcp", "127.0.0.1:8000")

if err != nil {

log.Fatal(err)


```
}

// Ensure server disconnect when done
defer conn.Close()

// Communicate with the server
for i := 1; i < 4; i++ { // Loop 3 times
    conn.Write([]byte(fmt.Sprintf("Client 2 sent message #%d\n", i)))
    time.Sleep(1000 * time.Millisecond)
}

conn.Write([]byte("Client 2 exiting\n"))
}

// Handle the client connection for the server
func connHandler(conn net.Conn) {
    // Defer, but guarantee, to close the client connection
    defer conn.Close()

    // Copy from conn to os.Stdout until EOF or error
    n, err := io.Copy(os.Stdout, conn)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println("Number of bytes received from client:", n)
    fmt.Println("Server connection handler exiting")
}

// Server goroutine
func simulateServer() {
    // Create a server connection
    svrconn, err := net.Listen("tcp", "127.0.0.1:8000")
    if err != nil {
        log.Fatal(err)
    }

    // Defer, but guarantee, to close the server connection
    defer svrconn.Close()
    for {
        // Accept connection from client request
        conn, err := svrconn.Accept()
        if err != nil {
            log.Fatal(err)
        }

        // Let connection handler deal with client
        go connHandler(conn)
    }
}
```

```
}

func main() {
    go simulateServer()    // Launch a server simulator
    go simulateClient1()   // Launch a server client simulator
    go simulateClient2()   // Launch a server client simulator
    // Give goroutines time to run
    time.Sleep(10 * (1000 * time.Millisecond))
    fmt.Println("Main exiting")
}
/*
Prints something like:
Client 2 sent message #1
Client 1 sent message #1
Client 1 sent message #2
Client 2 sent message #2
Client 2 sent message #3
Client 1 sent message #3
Client 2 exiting
Number of bytes received from client: 92
Server connection handler exiting
Client 1 sent message #4
Client 1 sent message #5
Client 1 exiting
Number of bytes received from client: 142
Server connection handler exiting
Main exiting
*/
```

Condition Variable

There are occasions when a goroutine might want to check on an external condition before performing an action. This is a situation where a program may use a condition variable. Rather than polling to check the state of the condition, it is more efficient if the goroutine goes to sleep in a wait state, to be awoken when the condition changes.

The general use case is when there are many workers available to perform a task or tasks, however sometimes no tasks are available for processing. In this case the workers need to wait. When tasks become available, the workers need to be informed. There are a couple of approaches to doing this in Go.

One way is to use channels, where the worker goroutines either block on a read on separate individual synchronous channels (one for each worker); or collectively block on a common buffered asynchronous channel. Channels provide the flexibility to avoid blocking on a read by using *select*. Channels also provide the ability to wait with a timeout by using a call to *time.After(duration)* which returns a channel containing a timestamp after the duration has elapsed; when used in combination with the *select* statement listening to another channel then the goroutine can wait for *time = duration* and then move on.

The other way is to use a common condition variable. Go provides the capability via the *sync* package which supports condition variables (as well as mutual exclusion locking, map functions, wait groups, and the pool capability for sharing data structures between goroutines). The four condition variable functions are: create a new condition variable via *NewCond()*, *Wait()* for a condition variable signal, *Signal()* a single goroutine waiting on a condition variable, and *Broadcast()* a signal to all goroutines waiting on a condition variable.

The condition variable is of type *Cond*, which is a struct containing a single field of type *Locker*. Now type *Locker* is an interface type with two methods: *Lock()* and *Unlock()*. So the condition variable has mutual exclusion locking. See [Mutex](#).

Here is the major difference between using condition variables rather than channels when notifying goroutines of a state change. With a condition variable the goroutines will block indefinitely until notified. When using channels, the goroutines have the option of waiting for a while and then breaking from the wait state so they do some processing, then going back to check on the channel for notification. See [Channels: Code Example](#) for an example.

The following code example shows two goroutines that block in a *wait* state on a condition variable inside a *for* loop. When the main goroutine sends a broadcast signal on that condition variable, the two goroutines wake up and commence processing. After processing, they loop back to *wait* again. This continues until the program exits.

Condition Variable: Code Example

```
// Illustrate condition variable usage
package main

import (
    "fmt"
    "sync"
    "time"
)

// Condition variable and send only channel
func mygo1(c *sync.Cond, ch chan<- string) {
    // Lock() to prevent panic of Wait() unlocking an unlocked mutex
    c.L.Lock()
    for {
        // Wait atomically unlocks c.L, suspends goroutine until signal received
        c.Wait() // After signal received, Wait locks c.L before returning

        // Signal received, done waiting
        fmt.Println("mygo1 received signal")

        // Simulate doing work
        time.Sleep(2000 * time.Millisecond)

        // Just have this here to let main know mygo1 is done
        ch <- "mygo1 did some work"
    } // Loop forever until program ends
}

// Condition variable and send only channel
func mygo2(c *sync.Cond, ch chan<- string) {
    // L is of type Locker which is interface with 2 methods Lock() and Unlock()
    c.L.Lock()
    for {
        // Wait atomically unlocks c.L, suspends goroutine until signal received
        c.Wait() // After signal received, Wait locks c.L before returning

        // Signal received, done waiting
        fmt.Println("mygo2 received signal")

        // Simulate doing work
        time.Sleep(1000 * time.Millisecond)

        // Just have this here to let main know mygo2 is done
        ch <- "mygo2 did some work"
    } // Loop forever until program ends
}

func main() {
```

A Cyclopedia of Go

```
// Only need channel to know goroutines have done work
ch := make(chan string) // Unbuffered synchronous channel

// NewCond returns a new Cond with Locker L.
c := sync.NewCond(&sync.Mutex{})

// Pass condition variable pointer and write only channel
go mygo1(c, ch)
go mygo2(c, ch)

// Let goroutines get into their wait loops
time.Sleep(100 * time.Millisecond)

// Send the signal to wake up all Wait in goroutines
c.Broadcast()

// Collect goroutine exit messages
fmt.Println(<-ch)
fmt.Println(<-ch)

// Let goroutines get into their wait loops
time.Sleep(3000 * time.Millisecond)

// Send the signal to wake up all Wait in goroutine
c.Broadcast()

// Collect goroutine exit messages
fmt.Println(<-ch)
fmt.Println(<-ch)
fmt.Println("Exiting...")
}

/*
Prints something like:
mygo2 received signal
mygo1 received signal
mygo2 did some work
mygo1 did some work
mygo1 received signal
mygo2 received signal
mygo1 did some work
mygo2 did some work
Exiting...
*/
```

Constants

Constant refers to something that is unvarying, not changeable. The word comes from Late Middle English and is derived from the Latin word “*constāre*” which means “to stand firm”.

Constants can only be declared as basic types: string literals, numbers, or booleans.

String literals can be assigned to both constants and variables. String literals are untyped string constants and are referred to as “immutable” in the Go language specification. See the [Immutability](#) topic.

Constants are expressions which are known at compile time. Constants cannot be assigned values that cannot be determined at compile time. Unlike variables, the compiler will not warn of, and but will permit, constants that are unused in a package.

Once constants are assigned a value, the value of the constant may never be changed.

Pointers may not be assigned to the address of a constant, the compiler forbids this.

Go does not have an enumerated type in the language. But a set of constants can be created as a sequence of enumerated values, where the “constant generator” called iota can be used to initialize the set of constants.

See the following for valid and invalid (commented out to compile) code:

Constants: Code Example

```
// Illustrate the use of constants
package main

import (
    "fmt"
    "math"
)

const a = "hello" // Valid operation
const b = 50      // Valid operation
var c = 100 // Valid operation: also scope is for whole package, does not need to be used

type yearInCollege int

const (
    _      yearInCollege = iota // 0 - assign to the “blank identifier” ‘_’ so 0 value is ignored
    freshman                // 1
    sophomore                // 2
    junior                   // 3
)
```

```
        senior          // 4
    ) // All constants in the constant generator are of type yearInCollege

type student struct {
    firstname string
    lastname  string
    year      yearInCollege
}

func main() {
    // a = "world" // Invalid operation: cannot assign new value to constant a
    // b = 25 // Invalid operation: cannot assign new value to constant b
    // const w = math.Sqrt(9) // Invalid operation: constant value not determined until run time

    const x = 1 // Valid operation: integer constant x is assigned an integer, never used
    y := "foo" // Valid operation: y is both declared and assigned a string constant value
    y = "bar" // Valid operation: variable y is reassigned to a different string literal
    var z = math.Sqrt(9) // Valid operation: variable assignment can be made at run time
    s := student{"Big", "Dog", senior} // The struct is initialized with 2 strings and a constant

    fmt.Println(y, z, s) // Variables must be used, constants do not have to be used

    i := 5
    j := 10
    p := &i // Valid operation: pointer can be assigned to address of variable
    fmt.Println(*p)
    p = &j // Valid operation: pointer can be assigned to another address of variable
    fmt.Println(*p)
    //p = &x // Invalid operation: pointer cannot be assigned to address of a constant
    //fmt.Println(*p)
}
/*
Prints the following:
bar 3 {Big Dog 4}
5
10
*/
```

Data Structures

A structure is an object composed of one or more parts assembled in a particular way, and is considered as a whole. The word originates from the Latin “*structūra*”, which means “to fit together”.

A data structure is a particular way of organizing data in computer memory. Each data structure is based either on a defined data type, or an aggregation of data types. The data types may be defined by the language or specified in a program. When specified in a program, the specified types may be based on the language defined types, or an aggregation of language defined and program specified types, or an aggregation of program specified types.

Data structures have limitations on the operations that can be performed upon them, and the functions and methods that may be applied to them. The operation limitations are inherent to the language defined types, and the function and method limitations are based on the signatures of those functions or methods.

All data structure elements exist within contiguous memory. But data structures composed of multiple elements may exist non-contiguously in the computer memory. For example, an individual struct is an aggregate data structure that exists within contiguous memory, but a linked list of structs is non-contiguous in the computer memory.

A pointer is a data structure that “points” to another data structure. The pointer data structure contains one element within it, this is the address location of another data structure within computer memory. At this time on a 32-bit machine, the size of the pointer itself will be 4 bytes; whereas on a 64 bit machine, the size of the pointer itself will be 8 bytes. The reason is that it must potentially reference a much larger memory space.

Following will be some graphic illustrations to show how data structures impact memory. At the end of this section will be four code examples showing how to derive the the information related to memory usage of various data structures.

Suppose it is specified that: “`x := [5]int16{1, 4, 9, 16, 25}`” and that “`y := &x`”. On a 32 bit machine, this will result in array x consuming 10 bytes (as type `int16` consumes only 2 bytes). Referencing the memory addresses and values would reveal something like the following:

Address	Meaning	Value	Bytes
0x10410020	Address of array x	[1 2 3 4 5]	10
0x10410020	Address of x[0]	1	2
0x10410022	Address of x[1]	4	2
0x10410024	Address of x[2]	9	2
0x10410026	Address of x[3]	16	2
0x10410028	Address of x[4]	25	2
0x1040c128	Address of pointer y	0x10410020	4

In the preceding table it is evident that the 20 bytes allocated for the array are contiguous, as seen by the trailing hexadecimal values in the address: 20, 22, 24, 26, 28. The address reference for the first cell in the array is the same as the address of the array itself. Finally, the pointer assigned to point to the array has the address of the array as its value. See [Data Structures: Code Example - Array and Array Pointer](#).

Another example will be to look at a linked list of structs that will also be loaded with the same five values. Examining the memory address will show that this data structure, while holding similar integer information, is scattered throughout the memory and is not contiguous.

Referencing the memory addresses and values of the linked list nodes would reveal something like the following:

Address	Meaning	Value	Bytes
0x10444260	Address of node	1	20
0x10444280	Address of node	4	20
0x104442a0	Address of node	9	20
0x104442c0	Address of node	16	20
0x104442e0	Address of node	25	20
0x1040c130	Address of pointer p	0x10444260	4

In the preceding table it is evident that the 20 bytes allocated for each linked list element node are not contiguous, as seen by the trailing hexadecimal values in the address: 260, 280, 2a0, 2c0, 2e0. (The addresses are hexadecimal, so the distance between each address is 32 bytes, not 20 bytes.) Finally, the pointer assigned to point to the first node in the linked list has the address of the first node as its value, and again the pointer is 4 bytes in size. See [Data Structures: Code Example - Linked List](#).

Interestingly, by using the package provided doubly linked list functionality, it does not matter if the assigned values are cast to type *int16* or *int64* before assignment, the node remains the same size. This is because if “*unsafe.Sizeof(e.Value)*” is performed, it will be revealed that the value is assigned into an 8 byte space. Because each node contains the value, a next pointer, a prev pointer, and a pointer to the list itself, that will be 8 + 4 + 4 + 4 bytes adding to 20 bytes. See the “*container/list*” package and follow the link to the “*list.go*” source code.

The struct data type is contiguous in memory (including internal padding), and structs may embed other structs. Are the embedded structs actually part of the embedding structs memory sequence, or is something else going on? To understand struct sizing in memory, there is both alignment and padding happening.

Two different structs with the same field types may have a different size due to the order of the fields in the struct. Suppose the following are declared: “*var a struct {a string; b bool; c bool}*”, “*var b struct {a bool; b string; c bool}*”, and “*var c struct{a bool; b bool; c string}*”. While all of these structs have the same sized fields, because they are in a different order in each struct, the structs are not all the same size. This is shown in the following graphic.

var a struct {a string; b bool; c bool} size in memory is 12 bytes (32 bit system)

b0	b1	b2	b3	b4	b5	b6	b7	string	0x1040a0b0
b8								bool	0x1040a0b8
	b9							bool	0x1040a0b9
		ba	bb					padding	0x1040a0ba

var b struct {a bool; b string; c bool} size in memory is 16 bytes (32 bit system)

c0								bool	0x1040a0c0
	c1	c2	c3					padding	0x1040a0c1
c4	c5	c5	c7	c8	c9	ca	cb	string	0x1040a0c4
cc								bool	0x1040a0cc
	cd	ce	cf					padding	0x1040a0cd

var c struct{a bool; b bool; c string} size in memory is 12 bytes (32 bit system)

d0								bool	0x1040a0d0
	d1							bool	0x1040a0d1
		d2	d3					padding	0x1040a0d2
d4	d5	d6	d7	d8	d9	da	db	string	0x1040a0d4

Padding is different on 32-bit systems than on 64-bit systems. On 32-bit systems, the padding is to the nearest four bytes, whereas on a 64-bit system, padding to the nearest eight bytes. So, on a 64-bit system, the sizes of the 3 structs would be 24, 32, and 24 bytes respectively. The default size on a 64-bit system of a string data type is 16 bytes, not 8 bytes. However, a *bool* is 1 byte regardless of whether it is a 32-bit or 64-bit system. The code that produces the displayed results is included in the data structures code examples section that follows. See [Data Structures: Code Example - Struct Field Ordering](#).

It is interesting to see what happens when a struct embeds a struct. The embedded struct is contiguous within the embedding struct. In the following example, the embedded struct has one field, which is a function type. As will be shown, function types consume 4 bytes in a 32 bit system. Note that the assigned string literal is not part of the struct itself, the string field merely points to the string literal. It is not possible to determine the address of a string literal, since it is a constant. Total size is $4 + 1 + 1 + 2 + 8 = 16$ bytes. See [Data Structures: Code Example - Struct Embedding Struct](#).

type a struct { f func(name string) string }									
var b struct{a; b bool; c bool; d string}									
e := b{b: true, c: true, d: "A great big very fat rabbit"}									
e.a = a{f: func(name string) string {return "Foo " + name}}									
b0	b1	b2	b3					func	0x1040a0b0
b4								bool	0x1040a0b4
	b5							bool	0x1040a0b5
		b6	b7					padding	0x1040a0b6
b8	b9	ba	bb	bc	bd	be	bf	string	0x1040a0b8

Following are the four code examples demonstrating the memory usage of these data structures.

Data Structures: Code Example - Array and Array Pointer

//Illustrate the impact on memory of an array and an array pointer

```
package main
```

```
import (
    "fmt"
    "unsafe"
)

// Show array contiguous addresses, pointer value is first array address
func a() {
    x := [5]int16{1, 4, 9, 16, 25}
    y := &x
    fmt.Printf("Address of array x: %p\n", &x)
    fmt.Printf("Value of array x: %v\n", x)
    for i, _ := range x {
        fmt.Printf("Address of x[i]: %p; Value of array x[i]: %v\n", &x[i], x[i])
        fmt.Println("Size of array cell:", unsafe.Sizeof(x[i])) // 2 bytes because int16
    }
    fmt.Printf("Address of pointer y: %p\n", &y)
    fmt.Printf("Pointer to array has address: %p\n", y)
    fmt.Printf("Pointer value is address: %v\n", y)
    fmt.Println("Size of pointer:", unsafe.Sizeof(y)) // 4 bytes pointer on 32 bit system
}

func main() {
    a() // Demonstrate array memory addresses and values
}

/* Prints something like the following:
Address of array x: 0x10410020
Value of array x: [1 4 9 16 25]
Address of x[i]: 0x10410020; Value of array x[i]: 1
Size of array cell: 2
Address of x[i]: 0x10410022; Value of array x[i]: 4
Size of array cell: 2
Address of x[i]: 0x10410024; Value of array x[i]: 9
Size of array cell: 2
Address of x[i]: 0x10410026; Value of array x[i]: 16
Size of array cell: 2
Address of x[i]: 0x10410028; Value of array x[i]: 25
Size of array cell: 2
Address of pointer y: 0x1040c128
Pointer to array has address: 0x10410020
Pointer value is address: &[1 4 9 16 25]
Size of pointer: 4
*/
```

Data Structures: Code Example - Linked List

```
// Illustrate the impact on memory of a linked list
package main

import (
    "container/list"
    "fmt"
    "unsafe"
)

// Show linked list has non-contiguous addresses, pointer value is address of first element
func ll() {
    // Create a new list and put some numbers in it
    l := list.New()
    fmt.Println("New list created, length:", l.Len())
    fmt.Println("Loading the list with 5 squares")
    // Load the list
    for i := 1; i < 6; i++ {
        l.PushBack(int16(i * i)) // Load with value of type int16
    }
    // Iterate through list and print its contents.
    for e := l.Front(); e != nil; e = e.Next() {
        fmt.Printf("Address of element: %p Value of element: %d\n", e, e.Value)
        fmt.Println("Size of list element:", unsafe.Sizeof(*e))
    }
    fmt.Println("List length now:", l.Len())
    p := l.Front()
    fmt.Printf("Pointer to list first element has address %p\n", &p)
    fmt.Printf("Pointer value is address %p\n", p)
    fmt.Println("Size of pointer:", unsafe.Sizeof(p)) // 4 bytes pointer on 32 bit system
}

func main() {
    ll() // Demonstrate linked list memory addresses and values
}

/* Prints something like the following:
New list created, length: 0
Loading the list with 5 squares
Address of element: 0x10444260 Value of element: 1
Size of list element: 20
Address of element: 0x10444280 Value of element: 4
Size of list element: 20
Address of element: 0x104442a0 Value of element: 9
Size of list element: 20
Address of element: 0x104442c0 Value of element: 16
Size of list element: 20
Address of element: 0x104442e0 Value of element: 25
```

```
Size of list element: 20
List length now: 5
Pointer to list first element has address 0x1040c130
Pointer value is address 0x10444260
Size of pointer: 4
*/
```

Data Structures: Code Example - Struct Field Ordering

```
//Illustrate the memory impact of struct field ordering
package main

import (
    "fmt"
    "unsafe"
)

func a() {
    var a struct {
        a string
        b bool
        c bool
    }
    fmt.Println("Size of struct a:", unsafe.Sizeof(a))
    fmt.Printf("Address of struct a: %p\n", &a)
    fmt.Printf("Address of struct a.a: %p\n", &a.a)
    fmt.Printf("Address of struct a.b: %p\n", &a.b)
    fmt.Printf("Address of struct a.c: %p\n", &a.c)
    fmt.Println("Size is 12, on a 64 bit system would be padded to 24")
}

func b() {
    var b struct {
        a bool
        b string
        c bool
    }
    fmt.Println("Size of struct b:", unsafe.Sizeof(b))
    fmt.Printf("Address of struct b: %p\n", &b)
    fmt.Printf("Address of struct b.a: %p\n", &b.a)
    fmt.Printf("Address of struct b.b: %p\n", &b.b)
    fmt.Printf("Address of struct b.c: %p\n", &b.c)
}

func c() {
    var c struct {
        a bool
        b bool
    }
```

```
        c string
    }
    fmt.Println("Size of struct c:", unsafe.Sizeof(c))
    fmt.Printf("Address of struct c: %p\n", &c)
    fmt.Printf("Address of struct c.a: %p\n", &c.a)
    fmt.Printf("Address of struct c.b: %p\n", &c.b)
    fmt.Printf("Address of struct c.c: %p\n", &c.c)
}

func main() {
    a()
    b()
    c()
}
/* Prints something like the following:
Size of struct a: 12
Address of struct a: 0x1040a0b0
Address of struct a.a: 0x1040a0b0
Address of struct a.b: 0x1040a0b8
Address of struct a.c: 0x1040a0b9
Size is 12, on a 64 bit system would be padded to 24
Size of struct b: 16
Address of struct b: 0x1040a0c0
Address of struct b.a: 0x1040a0c0
Address of struct b.b: 0x1040a0c4
Address of struct b.c: 0x1040a0cc
Size of struct c: 12
Address of struct c: 0x1040a0d0
Address of struct c.a: 0x1040a0d0
Address of struct c.b: 0x1040a0d1
Address of struct c.c: 0x1040a0d4
*/
```

Data Structures: Code Example - Struct Embedding Struct

```
//Illustrating the effect on memory of a struct embedding another struct
package main
```

```
import (
    "fmt"
    "unsafe"
)

func main() {
    type a struct {
        f func(name string) string
    }
    type b struct {
```



```
    a
    b bool
    c bool
    d string
}
e := b{b: true, c: true, d: "A great big very fat rabbit"}
fmt.Printf("Address of struct e: %p\n", &e)
fmt.Println("Size of struct e:", unsafe.Sizeof(e))
fmt.Printf("Address of struct e.a: %p\n", &e.a)
fmt.Printf("Address of struct e.b: %p\n", &e.b)
fmt.Printf("Address of struct e.c: %p\n", &e.c)
fmt.Printf("Address of struct e.d: %p\n", &e.d)
fmt.Println("Struct e.d has assigned string:", e.d)
e.a = a{f: func(name string) string {return "Foo " + name}}
fmt.Println("Struct e.a function call produces:", e.a.f("Bar"))
}
/*
Prints something like the following:
Address of struct e: 0x1040a0b0
Size of struct e: 16
Address of struct e.a: 0x1040a0b0
Address of struct e.b: 0x1040a0b4
Address of struct e.c: 0x1040a0b5
Address of struct e.d: 0x1040a0b8
Struct e.d has assigned string: A great big very fat rabbit
Struct e.a function call produces: Foo Bar
*/
```

Deadlock

A deadlock is a condition where progress is prevented, a stalemate, or impasse. It is derived from two words: Old English “*dēad*” meaning inanimate or inactive, and Old English “*loc*” meaning barrier or fastening.

For programming languages this refers to a situation where two or more processing threads are halted due to waiting for the other threads to complete an action. Most generally the cause is resource contention over access to shared resources that require use of locks to control access but can also be caused when threads are waiting a message to be received before performing an action and the message is never received.

The “Coffman Conditions” assert that there are four conditions required for a deadlock: resources are not shareable at time of access, acquired resources must be released by current owner to be accessed, each waiting thread has already acquired at least one resource from the set of required resources, and each thread is part of the set of waiting threads where the other threads already hold resources from the resource set.

For example, suppose there are two resources, and two processing threads. Both resources must be accessed to complete an action, and access is controlled by locking. Normal processing is for a thread to lock both resources, perform the action, and release both locks. A deadlock can occur when one thread acquires a lock on one of the resources, but the other thread acquires a lock on the other resource. At this point neither thread can proceed.

However, this definition is too restrictive. Another situation is where a thread has successfully locked access to a resource but then never releases the lock. In this case any other threads trying to access the resource are deadlocked.

In the context of Go, deadlock can also occur without explicit locking. For example, if the main goroutine is waiting on a read on a channel, but no other goroutines will write to the channel, then a deadlock is detected, and the program is terminated. This is true only for the main goroutine.

To prevent deadlocks due to resource contention between goroutines, the goroutines can use mutual exclusion locks. The mutual exclusion locks are used to ensure only one goroutine can access a resource at a time.

A read on an empty channel will block (potentially forever), so use the dual argument form of the channel read operation to prevent this situation if this is not desirable.

Deadlock can also occur when using wait groups. This situation happens if goroutines do not decrement the wait group when they exit. Assume the main goroutine has launched a large number of goroutines and waits for them to complete. Each goroutine launched must be associated by adding 1 to the wait group, and when each goroutine terminates the wait group

must be decremented by 1. When the wait group number is at zero, the main goroutine will cease to wait. But if every goroutine launched does not decrement the wait group upon exit, the main goroutine will hang. The runtime observes this condition and halts and exits the program.

To guarantee locks are unlocked and wait groups are decremented, use the `defer` keyword. It ensures that when the method completes, it will unlock the lock or decrement the wait group. This will happen whether the goroutine function completes normally, or whether it panics. In proper programming practice, within the goroutine make the call to `defer` immediately after acquiring a lock or after the wait group is incremented. Because `defer` guarantees that the resource will be unlocked or the wait group decremented even if the function panics, this will prevent deadlock.

For more examples of locks and wait groups, see [Mutex \(Mutual Exclusion\)](#).

To prevent deadlocks when using locks and wait groups, see this simple example:

Deadlock: Code Example

//Illustrate deadlock prevention via the use of mutual exclusion locks

```
package main

import (
    "fmt"
    "sync"
    "time"
)

type exclusiveAccess struct {
    sync.Mutex // This is known as an embedded lock
    s1 string
    w int
    x int
    y int
    z int
}

func (m *exclusiveAccess) withLockGo1(val string) {
    m.Lock()
    defer m.Unlock() // Can cause deadlock if this line is commented out
    m.s1 = val
    i := m.w
    m.w = i + 1
}

func (m *exclusiveAccess) withLockGo2(val string) {
    m.Lock()
    defer m.Unlock() // Can cause deadlock if this line is commented out
    m.s1 = val
```

```

    i := m.w
    m.w = i + 1
}

func (m *exclusiveAccess) noLockGo3() {
    i := m.x
    time.Sleep(1 * time.Second)
    m.x = i + 1
}

func (m *exclusiveAccess) noLockGo4() {
    i := m.x
    time.Sleep(1 * time.Second)
    m.x = i + 1
}

func (m *exclusiveAccess) waitGroupGo5(val int, wg *sync.WaitGroup) {
    wg.Add(1)
    defer wg.Done() // Can cause deadlock if this line is commented out
    m.Lock()
    defer m.Unlock() // Can cause deadlock if this line is commented out
    m.y = m.y + val // With locking prevents race condition
}

func (m *exclusiveAccess) waitGroupGo6(val int, wg *sync.WaitGroup) {
    wg.Add(1)
    defer wg.Done() // Can cause deadlock if this line is commented out
    i := m.z
    time.Sleep(1 * time.Second)
    m.z = i + val // No locking so race condition occurs
}

func main() {
    var wg sync.WaitGroup // Enable wait grouping
    var m exclusiveAccess // Enable mutual exclusion

    go m.withLockGo1("foo") // Locking
    go m.withLockGo2("bar") // Locking

    go m.noLockGo3() // No locks
    go m.noLockGo4() // No locks

    for i := 1; i < 21; i++ {
        go m.waitGroupGo5(i, &wg) // Locking and wait group
    }
    for i := 1; i < 21; i++ {
        go m.waitGroupGo6(i, &wg) // No locking and wait group
    }
}

```

A Cyclopedia of Go

```
time.Sleep(3 * time.Second) // Pause to give goroutines time to run
wg.Wait()
fmt.Println("m.w contains: ", m.w) // Will contain 2
fmt.Println("m.x contains: ", m.x) // Will contain 1 due to race condition
fmt.Println("m.y contains: ", m.y) // Final value 210 with locking, otherwise indeterminate
fmt.Println("m.z contains: ", m.z) // Indeterminate result usually way less than 210
fmt.Println("m.s1 contains: ", m.s1) // Will contain either foo or bar
fmt.Println("Completed")
}

/*
Prints the following if m.Unlock() or wg.Done() are commented out in any goroutine:
fatal error: all goroutines are asleep - deadlock!
Otherwise prints:
m.w contains: 2
m.x contains: 1
m.y contains: 210
m.z contains: Some number usually way less than 210 because race condition
m.s1 contains: foo or bar (depends on whether withLockGo1 or withLockGo2 grabbed the lock first)
*/
```

Declarations

To declare means to explain, announce, make clear; it comes from Middle English “*declaren*” which came from Latin “*dēclārāre*”. In computer science a declaration is a construct of a programming language that specifies the properties of an identifier. Here are the aspects of Go declarations:

- For every program file, the file is declared into a package where it belongs.
- If a declared name within a program file is capitalized, it is accessible outside of the package in which it resides. This is known as exporting the capitalized entity.
- If a name is declared outside the scope of a function within a file, then it is visible and accessible within all files inside a package.
- There are four major kinds of declarations: variables, constants, types, and functions.

Variables - declarations consist of variable name, type, and the expression setting of an initial value. The type may be omitted if the expression is provided, and the expression may be omitted if the type is present. Full form: “*var name type = expression*”. A short form declaration may exist only inside function scope as “*name := expression*”. The Go language does permit variable shadowing, which is when a variable occurs with the same name in an inner block scope as exists in an outer block scope. When this happens, the inner block has priority. Changes to a variable within an inner block have no effect on the same named variable existing in an outer block. This will be illustrated in the example code following this section.

Constants - all constants are basic types which are numbers, booleans, strings, or runes; the declaration is of the form “*const name = expression*”. As with variables, multiple constants may be initialized in the same declaration as “*const (name1 = expression1, name 2 = expression2)*”. Note: shadowing also works on constants.

Types - declarations are used to create new types called “named types”, the new types are based on existing types, and the format is: “*type name underlying-type*”. Type declarations may differentiate uses of an underlying type or specify complex types. Warning: if a new type is declared from an existing type, the new type does **not** inherit the methods of the existing type.

Functions - declarations consist of the name of the function, the input parameter names and types, a list of return value names and types, and the body of the function. When functions are methods, then the declaration also contains the “receiver” of the type that the method is associated with (since all methods must be associated with a type).

See: [Embedding](#), [Functions](#), [Method Set](#), [Package](#), and [Types](#).

Declarations: Code Example

```
//Illustrating the usage of declarations
package main
```

```
import (
```

```
        "fmt"
    )

/*
    Package level declarations require the first character be a Unicode capital letter
    and be declared outside of any function. Visibility is outside the package,
    that means any file importing the package can see and access the named values
*/

// Declaration of package level constant
const A = 10

// Declaration of package level variables
var B = 20
var Child struct { // Declare anonymous structure
    firstname, lastname string
    age          int
}

// Declaration of package level types
type Person struct { // Declaration binds type name (Person) to a type (struct)
    firstname, lastname string
    age          int
}
type Man Person // Declare type Man to be of type Person
type Woman struct{ Person } // Declare named Woman type struct, embed type Person inside type
Woman
type Planet string // All variables of this type used for planet names only

// Declaration of package level method
func (p *Person) Load(fn, ln string, a int) {
    p.firstname = fn
    p.lastname = ln
    p.age = a
}

/*
    File level declarations require first character be a lowercase letter
    and be declared outside of any function. Visibility is inside the package only,
    any file in the package has access to the named values
*/

// Declaration of file level constant
const a = 10
const (
    b = 20
    c = 30
)
```

```
// Declaration of file level variables
var d = 40
var (
    e, f = 50, 60
)

// Declaration of file level type
type distance float64 // All variables of this type used for distance calculations only

// Declaration of file level method
func (p *Person) loadNamesOnly(fn, ln string) {
    p.firstname = fn
    p.lastname = ln
}

func main() { // Declaration of function main is only visible inside this file
    fmt.Println("const a and var d: ", a, d)
    var d = 100 // Declaration overrides file level variable - variable shadowing
    const a = 100 // Variable shadowing technique also applies to constants
    fmt.Println("Variable shadowing const a and var d: ", a, d)

    var p Person // Declare p to be of type Person
    var m Man     // Declare m to be of type Man
    var w Woman   // Declare w to be of type Woman
    fmt.Println("See default values of p, m, and w: ", p, m, w)

    p.Load("Jack", "Strong", 25) // Assign values to p via package level method
    fmt.Println("Person has: ", p)
    p.loadNamesOnly("Bill", "Smith") // Assign values to p via file level method
    fmt.Println("Person now has: ", p)
    //m.load("Joe", "Baker", 32) // Error: declaring type T2 to be T1 does not get T1's methods!
    m = Man{"Joe", "Baker", 32}
    fmt.Println("Man has: ", m)
    w.Load("Susan", "Smith", 27) // Can call methods of embedded Person field
    fmt.Println("Woman has: ", w)
}
/*
Prints the following:
const a and var d: 10 40
Variable shadowing const a and var d: 100 100
See default values of p, m, and w: { 0} { 0} {{ 0}}
Person has: {Jack Strong 25}
Person now has: {Bill Smith 25}
Man has: {Joe Baker 32}
Woman has: {{Susan Smith 27}}
*/
```

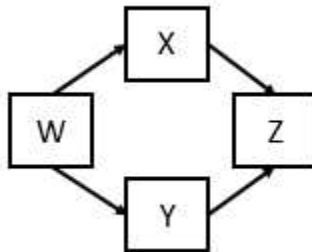

Dependency Management

Dependency management is a term used both for project management (where it refers to dependencies between tasks or activities) and for software management (where it refers to dependencies between software modules and between programs and their environments). Two terms for measuring dependency between software modules are *cohesion* and *coupling*, which are measures of how strongly or how weakly separate modules of functionality are connected with one another. The goal is high cohesion and loose coupling. Where functionality is strongly related it should be placed into a single package for high cohesion, and where functionality is weakly related or not at all related, then it should go into separate packages for loose coupling.

The interaction between packages should be implemented via interfaces. See [Interface](#).

In the context of Go, dependency management issues usually relate to package imports. The most basic issue with imports occurs when one package is dependent on another package, and the second package changes in a non-backwards compatible way such that when the first package imports the second package the program is broken.

A variation of this problem is known as the **diamond dependency** problem, and the problem can be illustrated thusly: Package W imports package X and package Y, each of which imports package Z. See illustration:



When package Z is compatible with both packages X and Y, there is no problem. But suppose package Z is modified in a non-backwards compatible way, and suppose package X requires the previous version of package Z, while package Y requires the current version of package Z. Now both versions of package Z are needed to resolve the problem.

(Another form of the diamond dependency problem occurs in languages that permit multiple inheritance for classes. Go does not support inheritance so that problem is avoided.)

Another import issue is called **cyclic import dependency**. It might occur like this. Suppose package W imports package X. No problems for W or X. Now suppose package X imports package Y. Still no problems for either W, X, or Y. But suppose package Y is updated and now imports package W. Any change to package W better not be made public because if it is, anytime any of the 3 packages are built they will encounter the error *"import cycle not allowed"*.

There is a general set of solutions to the import dependency problems. They are:

1. Don't build multiple projects in the same workspace
2. Perform vendoring on all packages sourced externally (vendoring means making a copy of an external package, and then referencing the copy)
3. Rewrite imports after vendoring

A reason not to build multiple projects in the same workspace is that separation more easily permits each project to depend upon different versions of vendored packages. This is also referred to as "GOPATH rewriting". Each project has their own GOPATH, and this is very easy to support when each project has its own workspace. Each project code set is completely isolated from any and all other project codesets.

External (vendor) packages are acquired by the *"get"* or *"install"* commands, and management of such packages depends upon the following conventions:

1. The import path depends upon the URL of the package, of the form "*<vendor domain name>/<vendor path to package>/<vendor package name>*".
2. Storage of the downloaded package in the local file system follows the form of the URL
3. Each directory under *"/src"* contains a single package
4. Each package is built using only information provided within its source code

After download, run testing to ensure the package does not break the program. If testing is successful, the package must be loaded into a source code control system for a project. Then use the package from the local source, rather than importing it from the external source. This eliminates the problem of a vendor package which is updated in a non-backwards compatible fashion from breaking a project's codebase. If a newly downloaded package breaks the program, overwrite it by pulling the good version from the source code control system. This overall process is known as "vendoring".

After a package is vendored, rewrite the import statement in packages that depend upon the vendored code to refer to the local version rather than the external version. This way the external version will never be accidentally imported for the codebase of the project.

Finally, when using open source 3rd party packages, it can sometimes make sense to simply extract the part of the functionality that is needed. This means not importing a huge library if only a small portion is needed. Copy the part that is needed into an internally managed package. Make sure you are conforming to the open source license when you follow this path.

There is no code example provided for this section, since the Go Playground does not support multiple packages. However, there are many examples of this feature on the Internet.

Embedding

In computer science the word “embedding” means to tightly enclose something within something else, mathematically it means to completely insert one set into another set. In the context of Go, embedding supports type composition via both data type embedding and interface embedding.

Consider that a struct is a sequence of zero or more fields, holding n named values of any type T . A struct may support data type embedding, as fields may be embedded within a struct, where an embedded field is an anonymous field that is a declared type. Since inheritance is not supported, when a struct B embeds a struct type A , B is not to be considered a subtype of A .

When one type is embedded into another type, the methods of the embedded type are available to the embedding type. When the methods are invoked, the receiver of the method is the embedded type, not the embedding type, but the embedding type has access. This is different than when a struct B is declared to be of type A , in this situation B does not have access to the methods of A .

For interfaces, embedding occurs in a very similar manner to data type embedding using structs. The embedding interface contains a field which is the anonymous type of the embedded interface. The method or methods of the embedded interface are accessible to the embedding interface; but when invoked the receiver of the method or methods is the embedded interface.

In the following code example the structure type “*person*” is embedded in the structure “*student*”. The method “*show()*” is a **promoted method** because it can be called by “*student*” even though it is a method owned by type “*person*”. Also, the fields in the embedded data type are directly accessible to the embedding type. They are considered **promoted fields** because they are accessed as if they were declared in structure “*student*” rather than in structure “*person*”.

The following code also shows the use of embedded interfaces. The interface *Show* embeds the interface *shower*. A call to the embedding interface *Show* demonstrates access to the embedded interface *shower* (which in turn maps to the method *show()*). See also [Composition](#), [Interface](#), [Method Sets](#), and [Polymorphism](#) for other code examples of embedding.

Embedding: Code Example

```
//Illustrating the principles of the concept of embedding
package main

import (
    "fmt"
)

type yearInCollege int
```

```

const (
    _      yearInCollege = iota // 0
    freshman // 1
    sophomore // 2
    junior    // 3
    senior    // 4
)

type person struct { // This is a named structure which creates a datatype person
    name string // Named explicitly
    age int     // Named explicitly
}

var student struct { // This structure is anonymous and is not a datatype
    person // Embedded anonymous field named implicitly by its type
    year   yearInCollege // Named explicitly, year is a constant
}

func (p person) show() string { // A person method
    return fmt.Sprintf("Name: %s, Age: %d", p.name, p.age)
}

type shower interface { // Interface satisfied by data type person's method
    show() string
}

type Show interface { // Interface embedding another interface
    shower
}

func main() {
    s := student // Variable s declared and initialized
    s.person = person{name: "Jack Frost", age: 19} // Assigning values to embedded type's fields
    s.year = sophomore // s.year assigned a constant value
    fmt.Printf("Name: %s, Age: %d, Student year: %d\n", s.name, s.age, s.year)
    fmt.Printf("%s, Student year: %d\n", s.show(), s.year) // Accessing embedded type's method
    var S Show // Declare S to be interface type Show
    S = s // Assign student to interface S
    fmt.Println("Call to embedding interface reveals: ", S) // Demonstrate embedding interface
}
/*
Prints the following:
Name: Jack Frost, Age: 19, Student year: 2
Name: Jack Frost, Age: 19, Student year: 2
Call to embedding interface reveals: {{Jack Frost 19} 2}
*/

```

Encapsulation

The word means to make or put into a capsule (which derives from Latin “*capsula*” which means box). In the context of computer science, it means a technique for the hiding of information. This is usually implemented a couple of different ways: the language enforces bundling methods with data, or the language restricts access to some components of an entity (in an object-oriented language, the entity is an object).

At a higher pattern level (above the programming language) encapsulation is common. For example, a RESTful service exposes its interface, and hides its implementation; as does a relational database which expose its query language interface but hides its internal structure.

In Go encapsulation is done at the package level. To hide variables, type methods, functions, or interfaces, declare them in lowercase. So, for example create a struct as a type, create methods that work on the fields of the type, and only expose certain capitalized methods outside of the package to manipulate the struct.

Go uses capitalization to enable encapsulation for variables, names of methods of types, function names, and the fields of structs. If the first letter of a variable is capitalized (Unicode uppercase letter) and outside of any explicit “{ }” scope binding (at the package block level), it is visible outside of its package.

Capitalizing the variable, function, type methods, or interface is known as *exporting*. See [Declarations](#). Note that a non-capitalized struct in a package can have a capitalized field which is accessible. See an example of this in [Package](#).

The general method for accessing package entities is via the interface. For example, suppose there is a package that exports an interface called *GenerateSound*. This interface has a method, called *makeSound()*. Suppose further there are two struct types called *Animal* and *Machine*, that both have their own *makeSound()* methods. *Animal* might be composed into *Dog* and *Cat*, while *Machine* may be composed into *Car* and *Truck*. Via the *GenerateSound* interface *Dog*, *Cat*, *Car* and *Truck* may all be requested to sound off: “bark”, “meow”, “beep”, and “honk” respectively. One exported (exposed) interface will handle all package entities that need to make sounds. See also [Composition](#), [Embedding](#), [Interface](#), and [Method Set](#).

Encapsulation: Code Example

```
// Illustrating basic encapsulation via use of an interface
package main

import (
    "fmt"
)

// Capitalized, visible outside of package
type Animal struct{ name string }
```

```
type Machine struct{ name string }
type Dog Animal
type Cat Animal
type Bulldozer Machine
type Jet Machine

// Methods lower case, not visible outside package
func (*Animal) makeSound() { fmt.Println("I am an Animal") }
func (*Machine) makeSound() { fmt.Println("I am a Machine") }
func (*Dog) makeSound() { fmt.Println("I am a Dog") }
func (*Cat) makeSound() { fmt.Println("I am a Cat") }
func (*Bulldozer) makeSound() { fmt.Println("I am a Bulldozer") }
func (*Jet) makeSound() { fmt.Println("I am a Jet") }

//Interface capitalized, visible outside package
type GenerateSound interface { // Interface
    makeSound()
}

func main() {
    var a Animal
    var m Machine
    var d Dog
    var c Cat
    var b Bulldozer
    var j Jet
    var GS GenerateSound

    GS = &a
    GS.makeSound()
    GS = &m
    GS.makeSound()
    GS = &d
    GS.makeSound()
    GS = &c
    GS.makeSound()
    GS = &b
    GS.makeSound()
    GS = &j
    GS.makeSound()
}

/*
I am an Animal
I am a Machine
I am a Dog
I am a Cat
I am a Bulldozer
I am a Jet
*/
```

Enumeration

The word is derived from Latin “*ēnumerātus*” which means number, enumerate with the “-ate” suffix means to make numbers (in a list); in computer science enumeration means creating an ordered listing of all items within a set of items.

An enumerated type in a programming language is a type containing a specific set of named values, and a variable that is declared to be a particular enumerated type may only be assigned values from the specific set designated for that enumerated type.

Go does not support the enumerated type in the language. Rather it permits the creation of a set of constants as a sequence of enumerated values. Suppose a variable can be declared to be of type “planet”, and then can be assigned any of the values mercury, venus, earth, mars, jupiter, saturn, uranus, neptune, pluto; as these are all constants of type planet. They have sequential and unique values due to the use of the iota generator. A typical use for this technique is to use the variable in a control flow switch statement where each case checks the variable against a constant, a case match means code can be executed.

Note that because the enumerated type is not supported, usage of enumeration permits the use of values beyond the range specified by the iota generator.

See the following:

Enumeration: Code Example

```
//Illustration of the use of a sequence of enumerated values
package main
import (
    "fmt"
)
type planet int
const ( // Use of iota constant generator for enumeration
    _      planet = iota // 0 (the _ means no assignment but iota does still increment)
    mercury // 1 (all constants in list will be of type planet, mercury == 1)
    venus   // 2
    earth   // 3
    mars    // 4
    jupiter // 5
    saturn  // 6
    uranus  // 7
    neptune // 8
    pluto   // 9
)

func porder(p planet) {
```

```
switch p {
case mercury:
    fmt.Println("Planet Mercury order counting out from the Sun: ", p)
case venus:
    fmt.Println("Planet Venus order counting out from the Sun: ", p)
case earth:
    fmt.Println("Planet Earth order counting out from the Sun: ", p)
case mars:
    fmt.Println("Planet Mars order counting out from the Sun: ", p)
case jupiter:
    fmt.Println("Planet Jupiter order counting out from the Sun: ", p)
case saturn:
    fmt.Println("Planet Saturn order counting out from the Sun: ", p)
case uranus:
    fmt.Println("Planet Uranus order counting out from the Sun: ", p)
case neptune:
    fmt.Println("Planet Neptune order counting out from the Sun: ", p)
case pluto:
    fmt.Println("Planet Pluto (yes I'm a planet) order counting out from the Sun: ", p)
default:
    fmt.Println("No such known planet")
}
}

func main() {
    var p planet // Variable p may be assigned any of the 9 same type constants
    p = earth
    porder(p)
    p = pluto
    porder(p)
    p = 11 // Because type planet is type int, this permissible
    porder(p)
}
/*
Prints the following:
Planet Earth order counting out from the Sun: 3
Planet Pluto (yes I'm a planet) order counting out from the Sun: 9
No such known planet
*/
```


Environmental Variables

All programs run in an environmental context that depends on the operating system. Environmental variables apply to the programs that run within that environment. Some of the information of interest to a program that may be provided by environmental variables include:

1. Directory path information, there are usually multiples of these.
2. Current directory location.
3. User profile information (all programs run “as a user”).
4. Setting the number of cores available to the program via GOMAXPROGS.
5. The name of the host server upon which the program runs.
6. Port names and numbers that are available for specific purposes.

Go has a package called “os” that has been designed and built to be platform independent, e.g. it should work from the perspective of the program the same way regardless of the operating system. In practice the package currently works for Linux, UNIX, Mac OS X, and Windows environments (with some limitations regarding operating system versions). It provides a number of functions that permit a program to interact with the operating system.

An example of a typical use of environmental variables is for a program to use these to determine where to access input files and where to write output files. This approach permits flexibility when the same program runs in different environments with different paths.

There are those who hold that it is more secure to use a configuration file that a program can access to load in this kind of information. In this approach either the path to the configuration file must be provided to the program as an argument when the program is executed, or the program must acquire the path to the configuration file from an environmental variable (the path to the configuration file or any other file should not be hard coded within a program).

Code example shows setting, unsetting, getting, and clearing all environmental variables. The call to “*os.Environ()*” lists all environmental variables.

Environmental Variables: Code Example

```
//Illustrating the usage of environmental variables
package main

import (
    "fmt"
    "os"
    "strings"
)

func main() {
    os.Setenv("FOO", "1") // Set an environmental variable
    os.Setenv("BAR", "foobar") // Set an environmental variable
    for _, e := range os.Environ() { //List all environmental variables.
```

```
    pair := strings.Split(e, "=")
    fmt.Println(pair[0], pair[1])
}
os.Unsetenv("BAR") // Unset environmental variable
fmt.Println("BAR:", os.Getenv("BAR"))
val, bool := os.LookupEnv("FOO") // Return value, if exists also return true, otherwise false
fmt.Println("Environmental variable FOO val contains:", val)
fmt.Println("Environmental variable FOO bool contains:", bool)
fmt.Println("Clearing all environmental variables")
os.Clearenv()           // Clear all environmental variables
val, bool = os.LookupEnv("FOO") // Return value, if exists also return true, otherwise false
fmt.Println("Environmental variable FOO val contains:", val)
fmt.Println("Environmental variable FOO bool contains:", bool)
}
/*
Prints the following:
FOO 1
BAR foobar
Unsetting environmental variable BAR
BAR:
Environmental variable FOO val contains: 1
Environmental variable FOO bool contains: true
Clearing all environmental variables
Environmental variable FOO val contains:
Environmental variable FOO bool contains: false
*/
```

Escape Analysis

The Go compiler performs escape analysis to determine whether variable memory space will be allocated in the heap area of memory or the stack area of memory. The determination is based on whether any given variable's memory allocation may escape the lexical scope of the context of its declaration. See [Heap](#), [Stack](#), and [Lexical Scope](#).

Wherever a function returns a reference to a variable that is declared within the lexical scope of the function, the memory space for that variable may be allocated on the heap. This is because if a function returns a pointer to a value where the address space of that value was allocated within the function, the calling program may manipulate the value pointed to by the pointer, and the address space of that value must be valid after the declaring function is popped off the stack.

However, the compiler might perform an optimization known as “inline expansion”. In this situation the compiler inserts the entire contents of the called function in the code location where the function would otherwise be called. When this occurs, the variable remains on the stack because no actual function call occurs. At the time of this writing the Go compiler can perform inline expansion on functions within the same package but will not perform inline expansion when calling functions imported from another package.

Wherever the compiler cannot determine whether a variable may be accessible outside of the scope of the function, the memory space for that variable is allocated on the heap.

Where a function declares a variable locally within its scope and does not return a reference to that locally declared variable, then the compiler determines that the variable does not escape from the lexical scope of that function. In this case the variable is allocated to the stack.

Note that saying “returns a reference” does not mean “pass by reference”. There is no passing by reference in the Go language, either coming into a function as arguments, or as values returned from the function. Everything passed is “pass by value”, e.g., copies of values for both input to a function and as output from a function. But a copy of a pointer will point to the same underlying value that the original pointer references, therefore if a function returns a pointer that points to a variable declared within the function, the variable pointed goes on the heap (unless inline expansion has occurred).

The passing of copies has an important implication. For example, suppose function *f()* calls function *g()*. Suppose *f()* passes variable *x* to *g(x)*, and *g(x)* returns the address of *x*, like “*return &x*”. Now *g(x)* declares *x* as an input parameter (as a copy of original *x*, being whatever type *x* in *f()* is, and holding whatever value *x* in *f()* has). Therefore, the pointer copy to *x* returned references the address of the copy of *x*. Unless this is inline expanded, that copy of *x* inside *g(x)* goes on the heap, even though it was an input parameter and not a variable with a variable declaration inside of *g(x)*.

The reason why the heap or stack determination is important has to do with program speed. Stack variable access is often an order of magnitude faster than heap variable access.

Heap memory is subject to memory fragmentation. Garbage collection (GC) does its best to clean up the heap periodically, but between GC cycles the heap memory becomes fragmented. This also slows down the program when it has to access noncontiguous memory areas within the heap. Finally, GC itself consumes processor time and periodically locks the program out from accessing the heap while GC is running. See [Garbage Collection](#).

To gain insight into the decisions of the compiler's escape analysis, use the "-gcflags" switch with the "-m" option when executing "go build". (Adding the "-l" flag prevents inlining.) Understanding the compiler's escape analysis decisions can aid in code optimization.

Escape Analysis: Code Example - Pass By Value

```
//Illustrating pass by value
package main

import (
    "fmt"
)

func f() (int, *int) {
    var x int
    x = 100
    fmt.Println("x and the address of x in f(): ", x, &x)
    fmt.Println("f() receives address of g() copy of x: ", g(x))
    return x, &x
}

func g(x int) *int {
    fmt.Println("g() receives receives a copy of x from f(): ", x, &x)
    return &x
}

func main() {
    v1, v2 := f()
    fmt.Printf("main() received output from f1(): %v %v\n", v1, v2)
}
/*
The function g() receives a copy of x from f(), it is stored in a separate location
The function f() returns its own copy of x and address of x to main()
Prints something like the following:
x and the address of x in f(): 100 0x416020
g() receives receives a copy of x from f(): 100 0x416028
f() receives address of g() copy of x: 0x416028
main() received output from f1(): 100 0x416020
*/
```

First Class Citizen

In a programming language, first class citizens are things that may be explicitly manipulated by a program at runtime. A first class entity must have three specific characteristics:

1. It must be able to pass lexical analysis, that is, it must have a valid syntactic value when the compiler applies the process of tokenization to the source code.
2. It must be able to have operators within the program applied to it. Keywords such as *for*, *else*, *if*, *range*, etc. are not first class entities.
3. It must be referenceable. That means that it must be accessible as an entity at runtime.

Variables, constants, and functions are all first class entities in Go. All may be manipulated by a program at runtime, all may have operators applied to them, and all are subject to lexical analysis if they are syntactically valid. They all may be pushed onto, and pulled from, the stack. First class citizens operations are: assignment to variables, passing to functions and methods, and returnable from functions and methods.

Because goroutines are functions, goroutines are also first class citizens.

Not all languages treat functions as first class entities. But in Go, functions may:

- Be given a name (binding an identifier to a value)
- Be assigned to variables
- Be members of data structures, such as structs, arrays, or members of linked lists
- Be passed to other functions
- Be returned from other functions

See the following example:

First Class Citizens: Code Example

```
// Illustrating usage of first class citizens (entities)
package main

import (
    "fmt"
)

func myf1() int { // Declare a function which returns integer value 2
    return 2
}

func myf2() int { // Declare a function which returns a function
    return myf1()
}
```

```
func myf3(i int) (x int) { // Declare a function which returns a named value
    x = i * i
    return // Because x is named value, don't have to do 'return x'
}

func main() {
    i := myf1() // i is first class citizen - operator applied, referenceable
    fmt.Println(i) // Can be passed to function
    fmt.Println(myf2()) // Function passed a function which returns a function
    j := myf3(i) // j is first class citizen, assigned function return value
    fmt.Println(myf3(j)) // Function with parameter passed to function as parameter
    fmt.Println(i + myf3(j)) // Operation can be applied to return value
}

/*
Prints the following:
2
2
16
18
*/
```

Functions

The word function derives from the Middle French “*fonction*” which comes from the Latin “*functionem*” meaning to perform or to execute. The type of a function specifies the set of all functions with the same signature, the signature describes all the function parameter and result types. Function facts:

- All parameters to a function are passed by value, e.g. they are copies
- All named result values are initialized to their type zero values on function entry
- All changes made to parameter values are contained within the function scope
- A function may update a value referenced by a pointer parameter value
- Functions can return multiple values
- Variadic functions can be called with varying numbers of the last parameter
- Named return values can be returned without being specified in a return statement
- A function declaration omits the body if it is external to the Go code (e.g. assembly)
- Functions may be passed to functions as parameters, and returned as values
- Functions may be assigned to pointers

A method is a function with a receiver; the method declaration binds the method (function) name to the method associated with the receiver’s type. See [Method Set](#).

An anonymous function is a **function literal**, which can be invoked directly or assigned to a variable. Function literals are known as closures and they may refer to and then enclose variables from the surrounding function (the function that the function literal is inside of). See [Literals](#).

The following example code shows various aspects of functions. These include the use of anonymous functions (see the assignments to variables *j* and *k*), and named functions *myf1()*, *myf2()*, and *myf3()*. It shows that function *myf1()* actually returns a function, that function *myf2()* is recursive, and that function *myf3()* is used as a goroutine. Finally it demonstrates the use of function composition (passing functions as parameters to functions, see assignments to *g* and *h*). See [Composition](#), [Channels](#), and [Goroutines](#).

Functions: Code Example

```
//Illustrating various aspects of functions
package main

import (
    "fmt"
)

func myf1(i int) func() int { // A function that returns a function
    return func() int { return 3 * i } // Anonymous function (function literal) returned
}
```

```

func myf2(i int) int { // Recursive function
    if i == 0 {
        return 1
    }
    return i * myf2(i-1)
}

func myf3(ch chan struct { // Function that will be called as goroutine
    string
    int
}) {
    defer close(ch) // Ensure will close channel no matter what
    for i := 1; i < 4; i++ {
        ch <- struct { // Write to buffered channel
            string // Will be assigned function result
            int     // Will be assigned result of operation
        }{fmt.Sprintf("String %d", i), i * 2}
    }
}

func main() {
    var i int
    var j func() int // Declare variable j to be of type function that returns int
    var k func(k int) int // Declare k to be of type function that takes an int and returns int
    i = 2
    j = func() int { return 2 } // Declaring anonymous function, assigning to variable
    k = func(i int) int { return 2 * i } // Declaring anonymous function, assigning to variable

    fmt.Println(j()) // Will print 2
    fmt.Println(k(i)) // Will print 4
    f := myf1(i) // Assigning result of named function to variable
    fmt.Println(f()) // Will print 6
    g := myf1(k(i)) // Pass result of call to function k() to function myf1()
    fmt.Println(g()) // Will print 12
    h := myf2(g()) // Assigning named result of named recursive function to variable
    fmt.Println(h) // Prints 479001600

    ch := make(chan struct { // Create buffered channel
        string
        int
    }, 2)
    go myf3(ch) // Goroutines are functions, pass the channel
    for k := range ch { // Loop with range clause iterates on buffered channel until closed
        fmt.Println(k.string, k.int) // fmt.Println() is a variadic function
    }
}

/*
Prints the following:

```


2
4
6
12
479001600
String 1 2
String 2 4
String 3 6
*/

Garbage Collection

In computer science, garbage collection (GC) refers to an automated method for releasing memory storage that contains out of date or invalid data. The resulting released memory is then available for use by the program. Because GC competes with the program code for accessing the memory space, there are times the GC has to halt program code, impacting performance

Go implements garbage collection via the runtime library linked into every Go program. The GC (version available as of the date of this document) is a concurrent, mark and sweep, tri-color collector. The Go GC is optimized for low latency. Latency is the time between a stimulus and an action, e.g. request and response, so in this context lowering latency means reducing the time that program threads are blocked from processing due to the running of the GC. The lower the latency, the less time threads are in a blocked state.

Tri-color means the heap is considered as a graph of connected data objects. At the beginning of each GC cycle all heap data objects are set to “white”. Next, each object referenced either globally or via stack pointers are set to “grey”. Then one by one, each “grey” object is set to “black” and checked to see if it points to any other objects. If the object pointed to is “white” it in turn is now set to “grey”. At the end of the process, all objects are either “white” which means they are not referenceable, or “black” which means they are. All “white” objects can now be removed from memory, and the memory can be released for program access. See [Heap](#).

Mark and sweep is really a scan, mark, and sweep. The scan phase scans the globals and the stack (or the multiple stacks if using goroutines) to collect the pointers to heap memory. The scan phase is “stop the world” (STW) where all program threads are halted but it is very fast. The mark phase identifies unreachable data objects and “marks” them for deletion. This phase runs concurrently with the program code, so the program is minimally affected by this GC phase. At the termination of the mark phase stack space is reclaimed and mark finalization occurs. Finally the sweep phase removes data objects and frees memory space.

Because the Go GC is implemented for low latency, it means the “stop the world” phase is very short. Since the GC runs concurrently with the program, most of the GC operation has minimal impact on threads. Running the GC in a multi-core environment means even less competition for core processing time. See [Parallelism](#) and [Concurrency](#).

The environmental variable GOGC controls the conditions under which GC activates. GC can actually be turned off by setting GOGC=off. Under rare conditions this may be desirable. The default value is GOGC=100, the number can be decreased or increased to affect GC timing.

The Go language is designed to let programmers optimize placing data storage on the stack rather than the heap. By avoiding heap storage, the GC processing time is reduced. Rather than trying to tune GC, a better approach is to tune the code by using the stack instead of the heap.

Generics

Generics in computer programming is a complex concept. Strictly speaking it is the plural form of the word generic, which means the condition of being applied to all members of a class of members. It is derived from the Latin term “*gener*” which is the stem of the Latin word “*genus*” and the Greek word “*génos*”, which means a kind, group, or a class, of entities. In the context of programming, the term is applicable to the situation where a function or an interface is expected to handle multiple different types as the same input parameter or parameters.

This is known as **parametric polymorphism** and may also be referred to as **parameterized types**. The programming languages that support parameterized polymorphism implement generics quite differently from one another. Go does not implement generics within the language because it does not do parameterized polymorphism. However, the programmer can instead handle generics via **ad hoc polymorphism** using **empty interfaces**, so technically rather than receiving multiple types of parameters the function or interface is always receiving the same type of parameters, which are of type interface.

The basic approach to generics is this: abstract the type from a procedure to enable the creation of a generic procedure that can handle multiple data types. For example, suppose there is a procedure that can “add” any two data types to each other: “add (a,b) {return a + b}”.

This means there will have to be a language convention to tell the generic procedure what the data types of the input parameters are. Because the function might be asked to add a complex number to a string, the language must be able to communicate to the function the data types of the parameters. The function will then be coded to handle the various scenarios. So *int + int* returns *int*, *int + float* returns *float*, *int + string* converts *int* to *string*, does concatenation, and returns *string*; and so on. This results in a single generic function that will vary in its operational actions based on the input data types (under the covers), while at a higher level the generic function is conceptually doing the same thing (in this example, “adding” two entities together).

Generics can be handled by using type *assertion* or the *reflect* package, or both. When using type *assertion*, the interface parameters are asserted to be of the desired type, which is needed so that addition or concatenation operations may be performed on them. See [Type Assertion](#).

Reflection is used to determine the underlying actual type of the values that were passed into the function or interface as parameters of type interface. The *reflect* package provides many functions, two of which enable determining a variable’s value and its type. Those two functions are *reflect.ValueOf()* and *reflect.TypeOf()*. They require that the variable passed into these functions is received as a parameter specified as “*interface{}*”. Whether the type passed in is a variable, a function, or an interface, it is received by these two functions as a parameter with type empty interface. See [Reflection](#).

The following code example shows a function that can take two parameters, each of which may be any type of *int*, *float64*, or *string*. Via the use of reflection and type assertion the function can

handle them all, effectively implementing the concept of “add (a,b) {return a + b}”. Since *ints* and *floats* cannot be added without type conversion, that is performed where required. And when adding an *int* or a *float* to a *string*, the number is converted to a *string* and the two *strings* are then concatenated. Finally, the function handles types which are not *ints*, *floats*, or *string* by using case *default*.

Generics: Code Example

```
// Illustrate the concept of generics using reflection and type assertion
package main
```

```
import (
    "fmt"
    "reflect"
)
```

```
// Add any two values if they are ints, floats, or strings in any combination, return an interface
// For simplification only float64 is handled
func myAdd(myUnknown1 interface{}, myUnknown2 interface{}) interface{} {
    var w int
    var x float64
    var y string

    // do reflection for type determination because parameters passed in as empty interfaces
    switch {
    // Case is both arguments are ints
    case reflect.TypeOf(myUnknown1) == reflect.TypeOf(w) && reflect.TypeOf(myUnknown2) ==
reflect.TypeOf(w):
        fmt.Printf("first argument is an int %v, second is an int %v\n", myUnknown1,
myUnknown2)
        return myUnknown1.(int) + myUnknown2.(int) // Type assertion

    // Case is both arguments are floats
    case reflect.TypeOf(myUnknown1) == reflect.TypeOf(x) && reflect.TypeOf(myUnknown2) ==
reflect.TypeOf(x):
        fmt.Printf("first argument is a float %v, second is a float %v\n", myUnknown1,
myUnknown2)
        return myUnknown1.(float64) + myUnknown2.(float64) // Type assertion

    // Case is both arguments are strings
    case reflect.TypeOf(myUnknown1) == reflect.TypeOf(y) && reflect.TypeOf(myUnknown2) ==
reflect.TypeOf(y):
        fmt.Printf("first argument is a string %v, second is a string %v\n", myUnknown1,
myUnknown2)
        return myUnknown1.(string) + myUnknown2.(string) // Type assertion

    // Case is first argument is an int, the second argument is a float
    case reflect.TypeOf(myUnknown1) == reflect.TypeOf(w) && reflect.TypeOf(myUnknown2) ==
reflect.TypeOf(x):
```

```

        fmt.Printf("first argument is an int %v, second is a float %v\n", myUnknown1,
myUnknown2)
        return float64(myUnknown1.(int)) + myUnknown2.(float64) // Type assertion and type
        cast

// Case is first argument is a float, the second argument is an int
case reflect.TypeOf(myUnknown1) == reflect.TypeOf(x) && reflect.TypeOf(myUnknown2) ==
reflect.TypeOf(w):
        fmt.Printf("first argument is a float %v, second is an int %v\n", myUnknown1,
myUnknown2)
        return myUnknown1.(float64) + float64(myUnknown2.(int)) // Type assertion and type
        cast

// Case is first argument is a number, the second argument is a string
case (reflect.TypeOf(myUnknown1) == reflect.TypeOf(w) || reflect.TypeOf(myUnknown1) ==
reflect.TypeOf(x)) && reflect.TypeOf(myUnknown2) == reflect.TypeOf(y):
        fmt.Printf("first argument is a number %v, second is string %v\n", myUnknown1,
myUnknown2)
        return fmt.Sprintf("%v", myUnknown1) + myUnknown2.(string) // Type assertion

// Case is first argument is a string, the second argument is a number
case reflect.TypeOf(myUnknown1) == reflect.TypeOf(y) && (reflect.TypeOf(myUnknown2) ==
reflect.TypeOf(w) || reflect.TypeOf(myUnknown2) == reflect.TypeOf(x)):
        fmt.Printf("first argument is string %v, second is number %v\n", myUnknown1,
myUnknown2)
        return myUnknown1.(string) + fmt.Sprintf("%v", myUnknown2) // Type assertion

// Case is any situation where either input parameter is not an int, float64, or string
default:
        fmt.Printf("first argument is %v, second is %v\n", myUnknown1, myUnknown2)
        return "Can only handle ints, floats, and strings"
    }
}

func main() {
    var (
        i1, i2, f1, f2, s1, s2, a1 = 2, 3, 3.14, 5.6432, "Foo", "Bar", [3]int{1, 2, 3}
    )
    fmt.Println(myAdd(i1, i2))
    fmt.Println(myAdd(f1, f2))
    fmt.Println(myAdd(s1, s2))
    fmt.Println(myAdd(i1, f2))
    fmt.Println(myAdd(f1, i2))
    fmt.Println(myAdd(f1, s1))
    fmt.Println(myAdd(s1, f2))
    fmt.Println(myAdd(s1, a1))
}

/*
```

Prints the following:

first argument is an int 2, second is an int 3

5

first argument is a float 3.14, second is a float 5.6432

8.7832

first argument is a string Foo, second is a string Bar

FooBar

first argument is an int 2, second is a float 5.6432

7.6432

first argument is a float 3.14, second is an int 3

6.14

first argument is a number 3.14, second is string Foo

3.14Foo

first argument is string Foo, second is number 5.6432

Foo5.6432

first argument is Foo, second is [1 2 3]

Can only handle ints, floats, and strings

*/

Goroutines

Go natively supports the concept of concurrent lightweight “threads” known as goroutines which run within the process space allocated to a Go program. An operating system thread may support many goroutine lightweight threads. These goroutines may be activated like calling a function prefixed by the keyword `go`. Unlike where calling a function the caller waits for the function to return, calls to goroutines return immediately. The main function itself is a goroutine and is known as the main goroutine. All goroutines are concurrent.

An example for the use of goroutines is a back-end service that may be required to handle many simultaneous connection requests. Each connection request will spawn a separate goroutine in the listener to handle that request. This functionality is covered by the “*net*” package. An example of the use of the “*net*” package may be found in the [Concurrency: Code Example](#).

Channels are used to facilitate communication between goroutines. Channels are created by a call to the *make()* function, and channels can be created as either buffered or unbuffered channels. The placing of data into a channel and removing data from a channel is done with the “<-” operator. Data flow is in the direction of the arrow.

It is useful to know that goroutines cannot pass pointers to one another that refer to memory within any goroutines stack. So, for example, goroutine *A* can declare an integer value and push it through a channel to goroutine *B* (which is sent as a copy). But goroutine *A* cannot declare an integer value and push a pointer to that value through a channel to another goroutine.

Calls to unbuffered channels block because these channels are synchronous. An example of communication via an unbuffered channel can be seen in the following code example. It demonstrates how one goroutine blocks after sending to the channel, and only unblocks after the other goroutine receives from the channel.

The other mechanism for communication between goroutines is via the use of mutual exclusion locks provided by the “*sync*” package. This can be a good mechanism when several goroutines need access to non-concurrent data structures such as maps, arrays, and slices.

In general, when using mutual exclusion locks the common practice is to permit multiple readers to access a given data structure, but writers have to queue up. The mutual exclusion locking system will only permit one writer at a time to have access to a data structure.

The following example is very simple, and just shows the independent execution of the goroutines. More detailed code examples can be found under [Concurrency](#), [Condition Variable](#), [Channels](#), and [Mutex \(Mutual Exclusion\)](#).

Goroutines: Code Example

//Illustration of Goroutines - very simple example

```
package main

import (
    "fmt"
)

func mygo1(ch1 <-chan int) { // Receive a read only channel
    <-ch1 // Block until receive value, discard it
    fmt.Printf("Goroutine 2 received value, discarded, exiting.\n")
}

func main() {
    var ch1 = make(chan int) // Unbuffered channel (synchronous)
    go mygo1(ch1)             // Launch goroutine
    fmt.Printf("Main goroutine sending to channel, blocking.\n")
    ch1 <- 12345              // Send to channel, block until received
    close(ch1)                // Close the channel
    fmt.Printf("Main goroutine closed channel, exiting.\n")
}
/*
Prints the following:
Main goroutine sending to channel, blocking.
Goroutine 2 received value, discarded, exiting.
Main goroutine closed channel, exiting.
*/
```


Heap

Heap refers to a collection or accumulation of items gathered in one place. The word comes from Old English “*hēap*”. In the context of computer science, it generally has two meanings. The first meaning relates to an area of memory reserved for a program that is not the stack.

The second meaning relates to a data structure that is a type of storage tree where the key value of any parent node is greater than any child node of that parent, and this is true for every subtree within the storage tree (for a “max” tree, the inverse is a “min” tree). Go has a package that implements the heap data structure, it may be accessed by importing “container/heap”. This type of heap will not be considered here.

Consider the first case, where the heap is dynamically reserved memory. In Go this allocation is performed for ordinary variable declarations when the compiler cannot determine whether a variable declared within a function will be referenced after the call to the function returns, or **sometimes** when the compiler determines that a function returns a reference to a variable declared within the function. This determination by the compiler is done when it performs escape analysis. If the compiler determines that a variable will only be referenced within a function, or will not be outside the function, then it is placed on the stack. See [Escape Analysis](#), also [Stack](#).

The *new()* function is for dynamic memory allocation of data structures that are not maps, slices, or channels; while *make()* is used to dynamically allocate maps, slices, and channels. A call to *make()* or to *new()* allocates space on the heap or on the stack. Note: the Go language specification does not indicate whether values will be created on the heap or on the stack; in fact, it does not reference either term. This is deliberate.

If it is desirable to know whether variables are placed on the heap or the stack, compile the program with the the gcflags argument *-m*. So: “*go run -gcflags -m myprog.go*”. To prevent inlining of functions, use the argument *-l* to verify whether an otherwise inlined function’s variable would be put on the heap or not.

Performance efficiency is improved when values can be restricted to the stack. However sometimes escape analysis allocates values to the heap. To handle these values Go has functionality called “garbage collection”. Periodically it sweeps through the heap and frees memory held by objects that are no longer accessible to the program. See [Garbage Collection](#).

Heap: Code Example

```
// Illustrate heap versus stack variables
// If an address of a variable created in f() is returned by f(), it may or may not be on the heap
// If the compiler can convert f() to an inline function, the variable will be on the stack
package main

import (
```

```

    "fmt"
    "math"
)

func f1(size float64) {
    var i int = int(size)
    buf := make([]byte, i) // buf on heap, because size of "i" is unknown when compiled
    fmt.Println(buf)
}

func f2() int {
    slice := make([]int, 10) // On stack, because never escapes f2() scope
    for i, _ := range slice { // Inserts sequential values into slice locations
        slice[i] = i + 1
    }
    var i int
    for _, n := range slice { // Adds total by summing all slice values
        i += n
    }
    return i // On stack because only a copy of value, not address of value, is returned
}

func f3() { // fptr in stack, not heap, because stays in f2() scope
    fptr := new(float32)
    *fptr = 3.15
    fmt.Println(*fptr)
}

func f4() *float32 { // Since f1() always returns *2, probably inlined
    var f float32
    f = 2.0
    return &f // If not inlined, then heap
}

func main() {
    var i float64 // On the stack
    i = 47.5
    size := math.Sqrt(i) // size has to be on the heap, compiler does not know value assigned
    fmt.Println(size)
    f1(size) // Nothing returned from this call
    fmt.Println(f2())
    f3() // Nothing returned from this call
    fmt.Println(*f4()) // Probably inlined to stack since function always returns the value 2
}

/*
Prints:
6.892024376045111
[0 0 0 0 0]

```

55
3.15
2
*/

Inheritance

Object Oriented languages usually support the concept of class inheritance, where a subclass of a parent class inherits the data entities and methods of the parent class, and then can extend them. Go does not support classes and does not support type inheritance. Instead the language provides a different approach.

The Go approach is as follows:

1. Any data type **may** have methods if they are non-virtual, if it does implement an interface method it can be referred to as an “interface type”
2. There is no “constructor” required, any data type declared (including complex types) will be initialized to default values if none are explicitly assigned
3. An interface is “similar to” part of a class, but it **only** has virtual methods, without data entities; because the methods are virtual this permits ad hoc polymorphism
4. A type “satisfies” an interface if it implements the methods of the interface; in this respect the type is also “similar to” part of a class in object-oriented programming languages
5. A type may be “embedded” into another type, if the embedded type has implemented methods then the embedding type has access to those methods; the design principle implemented by embedding one type into another is called composition. See [Composition](#).

The interface methods are external to the data type, and the data type is external to the interface. They are separate entities. Complex types have fields and may have only non-virtual methods (all struct methods are determined at compile time). Interfaces are types without fields, and with only virtual methods (thus enabling ad hoc polymorphism). Interfaces may be empty, in which case they are declared without any methods. Any type may satisfy the empty interface.

Combine the two together and the result is similar to (but not the same as) classes with inheritance, without the complexity and the overhead. There are some interesting advantages:

- Interfaces may be declared without methods (which can be added later)
- A data type can be declared without implementing methods, these can be added later
- A data type can add methods from multiple interfaces
- Different data types can implement the same interfaces

Bottom line: Go does **not** support inheritance. Go does **not** have classes. Go does **not** have subtypes. Instead it implements the design principles of encapsulation, composition and polymorphism via packages, embedding and interfaces. See [Composition](#), [Encapsulation](#), and [Polymorphism](#), also see [Package](#), [Embedding](#), and [Interface](#).

Interface

Interface means a common boundary between separate entities through which information may be shared. In Go an interface is a type. It is not a data type, it is an interface type. An interface is considered an abstract type because it is described by its behavior (via its methods), and the types that satisfy the interface may implement the methods. Because the methods of an interface contain no code, they are abstract. Go interfaces enable ad hoc polymorphism.

The zero value of an interface is “*nil*”. If no methods satisfy the interface, a call to the interface will cause a panic because “*nil*” does not reference a valid address in memory.

There is a special form of interface known as the empty interface, which has no methods. It looks like this: “*interface{}*”. A variable of type empty interface may have any value assigned to it. This technique is used when calling a function that must deal with unknown data types. In this situation the function should use double argument type assertion to validate the actual type. A code example for the empty interface is shown in [Type Assertion](#). Also see [Generics](#) and [Reflection](#).

A data type may call more than one interface. If a type has two methods in its method set, one method might satisfy one interface, and the other method might satisfy the other interface.

An interface may embed other interfaces. In this situation a variable of the type of the interface that has embedded other interfaces may call those other interfaces, if it is assigned to a variable that has methods that satisfy those other interfaces. So:

1. Suppose there is a datatype (say a struct) that has two methods e.g. *m1* and *m2*.
2. Each method of that type satisfies an interface e.g. *m1* satisfies *i1* and *m2* satisfies *i2*.
3. Suppose an interface *i3* embeds interfaces *i1* and *i2*.
4. Then a variable of interface type *i3* may be assigned to the data type which has methods *m1* and *m2*, and may call both methods named in interfaces *i1* and *i2*.

An example of an interface that embeds another interface may be seen in the topic on polymorphism. See [Polymorphism](#). See also [Method Set](#).

The following example code shows two separate complex data types (Book and Magazine). Each has methods to load the data type, and to display the contents of the data type. The display methods are identically named with an interface method. As such, each data type has “satisfied” the interface and may make use of it. Therefore, the Book and Magazine values may be assigned to the interface variable which can call the interface to show the values.

However notice the “*show()*” methods accept a receiver pointer. Therefore, the interface must also accept a receiver pointer. If a method accepts a type value, then the interface must receive a type value; if a method has a pointer receiver, then the interface must receive the address of the variable of the respective type. (A method with a pointer reference may be called either with a pointer to a value or with a value which has an address that is referenceable.)

Interface: Code Example

```
//Illustration of the use of an interface
package main
```

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

```
type Book struct{ author, title string }
```

```
type Magazine struct {
    title string
    issue int
}
```

```
// Book method set
func (b *Book) assign(n, t string) {
    b.author = n
    b.title = t
}
```

```
func (b *Book) show() {
    fmt.Printf("Author: %s, Title: %s\n", b.author, b.title)
}
```

```
// Magazine method set
func (m *Magazine) assign(t string, i int) {
    m.title = t
    m.issue = i
}
```

```
func (m *Magazine) show() {
    fmt.Printf("Title: %s, Issue: %d\n", m.title, m.issue)
}
```

```
// Declare interface, and methods that satisfy the interface
type shower interface { // By convention single method interface "-er"
    show()                // Interface only has one method
}
```

```
func main() {
    var b Book // Declare instance of Book
    var m Magazine // Declare instance of Magazine

    b.assign("Jack Rabbit", "Book of Rabbits") // Assign values to b via method
    m.assign("Rabbit Weekly", 26) // Assign values to m via method

    fmt.Println("Call data type methods")
}
```

```
b.show() // Show book values via method
m.show() // Show magazine values via method

var i shower // Declare variable of interface type
fmt.Println("Call interface")
i = &b       // Method has pointer receiver, interface does not
i.show()     // Show book values via the interface
i = &m       // Magazine also satisfies shower interface
i.show()     // Show magazine values via the interface
}

/*
Prints the following:
Call datatype methods
Author: Jack Rabbit, Title: Book of Rabbits
Title: Rabbit Weekly, Issue: 26
Call interface
Author: Jack Rabbit, Title: Book of Rabbits
Title: Rabbit Weekly, Issue: 26
*/
```

Immutability

Immutable refers to something that is unchangeable, unable to be changed. The word comes from Late Middle English and is derived from the Latin word "*immutabilis*".

Currently the only data type in Go that is immutable is type string. Immutability here refers to the value held in a distinct memory location. The term "string constant" refers to the value. Strings (the immutable memory area) may be assigned to variables. So, the **concept** of strings really refers to two distinct entities: the area in memory with the string value (this is the immutable part), and the area in memory of the string variable that points to the area of memory holding the string value. Multiple string variables may point to the same memory area containing the immutable string. String variables themselves are not immutable because they may be reassigned to point to different strings (the immutable parts). Because a string value is immutable, it cannot be modified.

See the code example following. First *w* is both declared and assigned a string value "foo". Then the variable *x* is assigned to point to the same immutable string value. Usually the assignment operator += will add the right operand to the left operand and assign the answer to the left operand, but it does not do this for operations on strings. Since strings are immutable, the operation must first allocate a new area of memory, then create a new string "foo bar" in that new location, and then switch the string address location of *w* from the old memory area to point to the new memory area. The result is two areas of memory where one area holds the string "foo" (pointed to by *x*), and a second area holding the string "foo bar", pointed to by *w*. The string variable *w* has also had its length changed from 3 to 7.

Because strings are immutable, they are concurrency safe. Multiple goroutines may access the same string safely, as no goroutine may make changes to a given string value.

Immutability: Code Example

```
// Immutability - show the effects of string literal assignments and concatenation
package main

import (
    "fmt"
)

type StructWithString struct {
    Val string
}

func (s StructWithString) setString(v string) string {
    s.Val = v
    return s.Val
}
```



```
func main() {
    w := "foo" // The w entity is variable, the "foo" entity is not variable and is immutable
    fmt.Println("w is", w)
    x := w // The x entity also now points to the immutable value "foo"
    fmt.Println("x is", x)
    w += " bar" // A new immutable string value is created: "foo bar"
    fmt.Println("w is now", w)
    fmt.Println("x is still", x)

    y := StructWithString{"whizbang"} // Create struct variable and load string field
    fmt.Println("y is", y)
    y.Val = "bangwhiz" // Assign new string to field variable, old string now inaccessible
    fmt.Println("y is now", y)
    y.setString("foobarwhizbang") // Variable y unchanged by function call because pass by value
    fmt.Println("y is still", y)
    z := y.setString("foobarwhizbang") // Function changes field value, assigned to z (y still same)
    fmt.Println("z is", z)
}

/*
Prints the following:
w is foo
x is foo
w is now foo bar
x is still foo
y is {whizbang}
y is now {bangwhiz}
y is still {bangwhiz}
z is foobarwhizbang
*/
```

Lexical Scope

Lexical means the vocabulary (words) of a language (lexicon means dictionary). It is derived from the Greek term “*lexikos*” which means “pertaining to words”.

Scope means a range of view, an extent; it is derived from the Latin term “*scopium*” meaning to look at, to view carefully. In the programming context it refers to the range of words over which an operator (such as “+”) has control; and the region of the program where the binding of a name to an entity is valid.

Lexical scope means that name **resolution** is dependent upon the location within the source code, and the **context** which is specified by where a variable or function is defined.

Scope in Go is managed in blocks, and these can be either explicit or implicit. Explicit blocks are contained within curly braces e.g. “{ }”. An explicit block can be empty (there may be nothing between the braces).

Implicit blocks are as follows:

1. Universal - all the Go source code in a program
2. Package - all the Go code in a package
3. File - all the Go code in an individual file
4. Logical - each “*for*”, “*if*”, “*else*”, and “*switch*” statement, and each clause in “*switch*” and “*select*” statements

All predeclared identifiers belong to the universe block scope. These are the language defined types, the constants “*true*”, “*false*”, and “*iota*”, the zero value “*nil*”, and finally a number of predeclared functions such as “*len()*”, “*make()*”, “*new()*”, and “*println()*”.

When a variable is passed to a function, what is passed is a copy of the variable. Within the function, the scope is applied to the copy. Any changes made to the copy will not be applied to the original variable.

When passing a pointer to a function, the passed pointer is a copy of the original pointer, but the pointer copy points to the same memory address. If within the function the pointer is reassigned to point to some other variable, when the function call returns the original pointer still points to the original variable. However, the function can modify the value of the variable to which the passed in pointer points. This is because the variable pointed to by the pointer retains its scope, whatever that is (package, file, curly braces, or logical e.g. “*for*”, “*if*”, “*else*”, “*switch*”, or clause in a “*switch*” or “*select*”).

The concept of using the same name for a variable within different blocks of scope is called “variable shadowing”. This means within each inner block of scope, a variable with the same name as a variable in an outer block of scope, is a different variable.

See the following code illustrating the concept of lexical scope, blocks, and variable shadowing:

Lexical Scope: Code Example

```
// Showing some aspects of lexical scope
package main

import (
    "fmt" // Scope of imported package name is file importing the package
)

const a = "A string literal" // Scope of constant is the package
var b = 100                  // Scope of variable is the package

func f1(p *int, x int) (i int) { // Scope of function is the package
    x++ // Scope is inside f1()
    fmt.Println("Inside f1() variable x incremented to: ", x)
    i = x
    *p = 20 // Changing the value of variable pointed to by pointer p
    fmt.Println("Inside f1() the variable p points to, is set to: ", *p)
    return // Do not need to say "return i" because i is named variable
}

func main() {
    x := 5 // Scope: main goroutine
    y := 10 // Scope: main goroutine
    z := 15 // Scope: main goroutine
    fmt.Println("x, y, and z in main goroutine block ", x, y, z)
    for x := 1; x < 3; x++ { // The variable x is shadowed in for block
        fmt.Println("x shadowed in for loop block, y and z still in main block ", x, y, z)
    }
    fmt.Println("x, y, and z in main goroutine block ", x, y, z)
    if y := "[y is now a string]"; len(y) == 9 {
        fmt.Println("x and z still in main block, y shadowed in if block ", x, y, z)
    } else {
        fmt.Println("x and z still in main block, y shadowed in else block ", x, y, z)
    }
    fmt.Println("x, y, and z in main goroutine block ", x, y, z)
    { // Demonstrates scoping within curly brace
        x := 100
        y := 200
        fmt.Println("x and y in unnamed {} block ", x, y)
    }
    fmt.Println("x, y, and z in main goroutine block ", x, y, z)
    p := &z
    y = f1(p, x)
    fmt.Println("Variable y set to return value from f1() ", y)
    fmt.Println("x, y, and z in main goroutine block ", x, y, z)
}
```

```
/*  
Prints the following:  
x, y, and z in main goroutine block 5 10 15  
x shadowed in for loop block, y and z still in main block 1 10 15  
x shadowed in for loop block, y and z still in main block 2 10 15  
x, y, and z in main goroutine block 5 10 15  
x and z still in main block, y shadowed in else block 5 [y is now a string] 15  
x, y, and z in main goroutine block 5 10 15  
x and y in unnamed {} block 100 200  
x, y, and z in main goroutine block 5 10 15  
Inside f1() variable x incremented to: 6  
Inside f1() the variable p points to, is set to: 20  
Variable y set to return value from f1() 6  
x, y, and z in main goroutine block 5 6 20  
*/
```

Linked Lists and Slices

A list is a series of names or items written in a sequence, the word is derived from Old High German “*leiste*”, and French “*liste*”. The original context was that “the lists” was a tournament field for competitive combat arts. To enter the tournament, a contestant had their name entered “in the lists”, where a list was the written series of names of contestants. In modern terminology, a list is any sequentially ordered written collection of related items.

In programming a linked list is a sequentially linked collection of elements, but they do not have to be adjacent in memory. A single linked list can only be traversed in one direction, from the head to the tail. A doubly linked list may be traversed in both directions. A linked list may be formed into a ring, in this case the list tail is linked to the list head; this is similar to a ring buffer. However, ring buffers use contiguous memory, and linked lists do not.

Because the items in the list are not in contiguous memory, it is easy to insert and remove items without having to deal with explicitly resizing the overall data structure. The items in the list are conventionally referred to as nodes, and the nodes are implemented as structs. The structs contain element information, as well as pointers to connect each struct to one another. The primary advantage is that memory does not need to be allocated in advance for the entire list, this is associated with the disadvantage that the linked list list nodes are scattered around in memory. Each node access requires following a pointer reference, as the structs are linked by pointers.

An alternative to a linked list is a dynamic array, which is implemented in Go as a slice. Slices use an underlying array, or pointer to array, or a string, or another slice. Unlike an array which is permanently fixed in size at compile time, slices may be increased in size at runtime.

Although slices are not limited to underlying array types, they are typically initially defined by being assigned to an array, or an existing slice assigned to an array. Slices can grow in size beyond the initial underlying array size by using the built in *append()* function, which appends zero or more values to an existing slice. When a slice does not have enough memory to add any more data, the append function allocates a new larger contiguous memory area and assigns the slice variable to the new memory area.

A danger of dynamic arrays in general, and using the *append()* function with slices in particular, is that if there is insufficient contiguous memory available for allocation the program will panic with an out of memory condition, and the program will terminate. So, while it is faster to traverse a slice using the *for* statement with the *range* clause than it is to sequentially follow the pointer references in a linked list, the linked list will (almost certainly) not run into an out of memory condition that can crash the program. Also, a doubly linked list can be traversed in both directions, the *for* statement with the *range* clause will only traverse in one direction.

If there are a truly large number of data elements, a data structure other than either a linked list or a slice is preferable. And when not dealing with a truly large number of data elements, slice

operations are faster than dealing with a linked list. Because slices can contain structs, it is generally preferable in Go to use slices and perform slicing than to use a linked list.

The language does provide a package with built in doubly linked list capability, to use this capability import the “*container/list*” package. It has built in functionality for list creation, addition and deletion of list members, movement of list members, and list traversal. An example of this may also be found in [Data Structures: Code Example - Linked List](#).

The following code example will demonstrate operations using a simple linked list, and then demonstrate how similar functionality can be achieved using slice operations.

Linked Lists and Slices: Code Example

```
//Demonstrate linked list and slice similar functionality
package main

import (
    "container/list"
    "fmt"
)

func ll() {
    fmt.Println("Demonstrate basic linked list functionality")
    // Create a new list and put some numbers in it
    l := list.New()
    fmt.Println("New list created, length:", l.Len())
    fmt.Println("Loading the list with 5 squares")
    // Load the list
    for i := 1; i < 6; i++ {
        l.PushBack(i * i)
    }
    // Iterate through list and print its contents.
    for e := l.Front(); e != nil; e = e.Next() {
        fmt.Printf("%d ", e.Value)
    }
    fmt.Printf("\n")
    fmt.Println("List length now:", l.Len())
    // Remove item from list
    for e := l.Front(); e != nil; e = e.Next() {
        if e.Value == 9 {
            fmt.Println("Removing", e.Value)
            l.Remove(e)
            break
        }
    }
    // Iterate through list and print its contents.
    for e := l.Front(); e != nil; e = e.Next() {
```

```

        fmt.Printf("%d ", e.Value)
    }
    fmt.Printf("\n")
    // Move item to end of list
    for e := l.Front(); e != nil; e = e.Next() {
        if e.Value == 4 {
            fmt.Printf("Moving %d to end of list\n", e.Value)
            l.MoveToBack(e)
            break
        }
    }
}
// Iterate through list and print its contents.
for e := l.Front(); e != nil; e = e.Next() {
    fmt.Printf("%d ", e.Value)
}
fmt.Printf("\n")
// Insert new item in front of the list
e1 := l.Front()
fmt.Println("Putting new value at front of the list")
l.InsertBefore(36, e1)
// Iterate through list and print its contents.
for e := l.Front(); e != nil; e = e.Next() {
    fmt.Printf("%v ", e.Value)
}
fmt.Printf("\n")
// Initialize (clear) the list
l.Init()
fmt.Println("Initialized (cleared) the list, length now:", l.Len())
}

func s() {
    fmt.Println("Demonstrate basic slice functionality")
    // Create a new slice and put some numbers in it
    s := make([]int, 0)
    fmt.Println("New slice created, length:", len(s))
    fmt.Println("Loading the slice with 5 squares")
    // Load the slice
    for i := 1; i < 6; i++ {
        s = append(s, (i * i))
    }
    // Print contents of slice
    fmt.Println(s)
    fmt.Println("Slice length now:", len(s))
    // Remove item from slice
    for i, e := range s {
        if e == 9 {
            fmt.Println("Removing", e)
            s = append(s[:i], s[i+1:]...)
            break
        }
    }
}

```

```

    }
}
fmt.Println(s)
// Move item to end of list
for i, e := range s {
    if e == 4 {
        fmt.Printf("Moving %d to end of slice\n", e)
        s = append(s[i:], s[i+1:]...) // Remove the 4
        s = append(s, e)              // Add 4 to end of slice
        break
    }
}
fmt.Println(s)
fmt.Println("Putting new value at front of the slice")
s = append(s, 0)
copy(s[1:], s[0:])
s[0] = 36
fmt.Println(s)
// Initialize (clear) the slice by setting it to nil
s = nil
fmt.Println("Initialized (cleared) the slice, length now:", len(s))
}

func main() {
    ll() // Demonstrate linked list functionality
    fmt.Println("+++++")
    s() // Demonstrate similar slice functionality
}

/*
Prints the following:
Demonstrate basic linked list functionality
New list created, length: 0
Loading the list with 5 squares
1 4 9 16 25
List length now: 5
Removing 9
1 4 16 25
Moving 4 to end of list
1 16 25 4
Putting new value at front of the list
36 1 16 25 4
Initalized (cleared) the list, length now: 0
+++++
Demonstrate basic slice functionality
New slice created, length: 0
Loading the slice with 5 squares
[1 4 9 16 25]
Slice length now: 5

```


Removing 9

[1 4 16 25]

Moving 4 to end of slice

[1 16 25 4]

Putting new value at front of the slice

[36 1 16 25 4]

Initialized (cleared) the slice, length now: 0

*/

Literals

A literal is a term used in computer science to denote a fixed value in program source code. Literal means the exact, precise meaning of an entity, that entity can only be what it is and no other thing. The word comes from Middle English and is derived from the Latin word *“litterālis”* which means “of letters”.

Variables and constants are not literals. Rather they contain or point to things that are literals.

Literals are normally stored in UTF-8 encoding format as code points. For example, a string literal expressed in the text of the Go program code such as “abc 表 😊” will be stored in UTF-8 format when the program is compiled. Numbers are also literals, and the compiled code expresses them in UTF-8 format.

Note: there is a rare exception to this situation. It is possible to deliberately escape the UTF-8 encoding format for strings by using escape characters within the quoted text of the string to specify an alternative encoding format. In this case the string will store the escaped characters as a sequence of bytes not following the UTF-8 format. This situation will rarely be encountered, because input data can and should be converted to UTF-8 format upon input (when required), and string literals written within the program itself will normally be written containing characters “as they are” in whatever language those characters are expressed. See [Runes](#).

Function literals are also known as anonymous functions and inline functions. They are written like ordinary functions declarations, but without a function name following the *“func”* keyword. They are expressions, not declarations. They have a property known as closure, meaning they can access, update, and remember variables declared in the enclosing function. See [Functions](#).

Anything that is not a keyword, an identifier (variables and types), or an operation or punctuation mark, is a literal. Almost all literals consume storage space, even when they are empty. There is one exception, which is if a type is specified as the empty struct, and an empty struct literal of that type is assigned to a variable, the variable will consume no memory space. This applies to complex types as well, so for example if an array is declared to hold values of empty struct, the array will also consume no memory space.

Struct literals have some special rules. These include: if no keys are listed, there must be a value provided for each field in the order of the fields, if a key is listed then all values must be preceded by keys, keys must be the field names of the struct, when keys are provided values are not required (values will default to the zero value for that field type), and finally the struct literal may be empty in which case all values will default to the zero values for the field types.

The following code shows a function literal and multiple kinds of literals assigned to variables. It also demonstrates how a struct literal of type empty struct consumes zero storage.

Literals: Code Example

```
// Literals - shows some examples
package main
```

```
import (
    "fmt"
    "unsafe"
)

func myf1() int {
    i := 1 // Following function literal acquires and encloses i
    return func() int { return 2 * i }() // Anonymous function (function literal)
}

func main() {
    type gender rune // Create rune type
    const (          // Create rune constants, all constants are literals
        male  gender = 'M'
        female gender = 'F'
    )
    type person struct { // Create struct type
        name string
        age  int8
        sex  gender
    }
    type empty struct{} // Create empty struct type
    var p person // Declare instance of struct type
    p.name = "James" // Variable p is not a literal, but it has field values containing literals
    p.age = 50
    p.sex = male // Assign values to struct fields
    q := person{name: "Jack", age: 20, sex: male} // q is not a literal, but has struct literal values
    r := person{"Jane", 30, female} // Another way to create struct literal values
    s := person{} // Demonstrates a struct literal with no values
    a := empty{} // Empty struct literal of type empty struct
    fmt.Println("Individual field values are literals", p)
    fmt.Println("Struct literal", q)
    fmt.Println("Another struct literal", r)
    fmt.Println("Struct literal with no values", s)
    fmt.Println("Size of struct literal with no values", unsafe.Sizeof(s)) // Shows storage allocated
    fmt.Println("Empty struct literal", a)
    fmt.Println("Size of empty struct literal with no values", unsafe.Sizeof(a)) // Shows zero storage
    fmt.Println("myf1 returns", myf1()) // Call function containing a function literal
    t := [2]string{"foo", "bar"} // An array literal
    fmt.Println("Array literal", t)
    fmt.Println("Size of array literal", unsafe.Sizeof(t))
    b := [2]struct{}{} // An array literal of empty structs
    fmt.Println("Array of empty structs literal", b)
    fmt.Println("Size of array literal of empty structs", unsafe.Sizeof(b)) // Shows zero storage
```

```
u := []string{"f", "o", "o", "b", "a", "r"} // A slice literal
fmt.Println("Slice literal", u)
v := u[2:4] // A new slice referencing part of other slice
fmt.Println("New slice", v)
w := map[string]float32{"foo": 12.34, "bar": 56.78} // All map literals require key values
fmt.Println("Map literal", w)
x := 100 // Variable x is not a literal, but the value 100 is a literal
fmt.Println("Integer literal", x)
y := 3.1416 // Variable y is not a literal, but the value 3.1416 is a literal
fmt.Println("Float literal", y)
z := "foobar" // Variable z is not a literal, but the value "foobar" is a literal
fmt.Println("String literal", z)
}

/*
Prints the following:
Individual field values are literals {James 50 77}
Struct literal {Jack 20 77}
Another struct literal {Jane 30 70}
Struct literal with no values { 0 0}
Size of struct literal with no values 16
Empty struct literal {}
Size of empty struct literal with no values 0
myf1 returns 2
Array literal [foo bar]
Size of array literal 16
Array of empty structs literal [{ }]
Size of array literal of empty structs 0
Slice literal [f o o b a r]
New slice [o b]
Map literal map[foo:12.34 bar:56.78]
Integer literal 100
Float literal 3.1416
String literal foobar
*/
```

Logic

The word logic means the ability to distinguish true from false reasoning and is derived from the Old French “*logique*”, which came from the Latin “*logica*” and Greek “*logike*”. Logical evaluation statements will return either true or false. Bivalent logic (the law of bivalence) asserts that every declarative statement expressing a proposition is either true or it is false. All of programming consists of statements of control flow, logical evaluation, or data operations. This applies throughout programming logic and is used in *if/else*, *for*, and *switch/case* statements.

Formal logic may be propositional only, or also predicate logic, and predicate logic consists of several orders or levels. A proposition is a statement describing a proposed state, and the proposition may be true, or it may be false. Propositional logic deals only with whether propositions are true or false. Propositions may also contain connectors linking subjects, and there are five possible connectors in propositional logic. These are: conjunction (and), disjunction (or), negation (not), implication (if/then), and equivalence (is, ==).

Here are some examples of propositional logic:

- Fast Rabbit **is** fast (equivalence)
- Slow Rabbit **is not** fast (equivalence, negation)
- Fast Rabbit **is** fast **and** Slow Rabbit **is** slow (equivalence, conjunction)
- **If** ((Fast Rabbit **is** fast **and** Slow Rabbit **is** slow) **and** (Fast Rabbit **and** Slow Rabbit run a race together)) **then** Fast Rabbit will win (implication, equivalence, conjunction)

Propositional logic (also known as zeroth order logic) contains predicates but does not perform quantifying logic on the predicates. Quantifiers specify quantities, such as *none*, *some*, *there exists at least one*, and *for all*. The word predicate means to assert or declare and comes from the Latin word “*praedicātus*”. Since a predicate is an assertion, looking at the proposition “*Fast Rabbit is fast*” then “*Fast Rabbit*” is the subject and “*is fast*” is the predicate assertion. In the proposition that “*Fast Rabbit lives in a rabbit hutch*”, “*Fast Rabbit*” is the subject, and “*lives in a rabbit hutch*” is the predicate. And the proposition is either true, or it is false, depending on the values composing the predicate.

Propositional logic does not permit variables, nor does it permit asserting the existence of an entity. So, while it is possible to form a proposition that “this car is blue” or “this car is not blue”, it is not possible to propose that “this car is *x*”, or to propose that “this *x* is blue”, where *x* is variable. It is also not possible to propose “there exists this car which belongs to a particular set of cars”. Therefore, zeroth order propositional logic is insufficient to support a programming language. Finally, propositional logic does not handle syllogistic arguments. For example, propositional logic cannot handle: (1) “All rabbits have fur”, and (2) “Fast Rabbit is a rabbit”, therefore (3) “Fast Rabbit has fur” because it cannot handle the “therefore” (as a logical consequence) logic.

First order logic handles existential quantification (there exists) and universal quantification (for all) applied to terms (entities) but does not permit quantification to be applied to properties of entities, relations between entities, and functions applied to entities.

An example within the context of universal quantification is that when ranging over an array of integers [1, 2, 3, 4] and those values are sequentially assigned to x , it is a universally true statement that for all x , $x^2 \geq x$.

In the context of existential quantification, when ranging over an array of integers [1, 2, 3, 4] and those values are sequentially assigned to x , it is an existentially true statement that there exists an x where $x^2 = x$, and that is $1^2 = 1$.

In all cases in first order logic, the quantification can only be applied to an individual entity even though the quantification is done over a domain of such entities e.g. *for all* x within a domain of entities, or *there exists* an x within a domain of entities. It can express statements like “if ($x == true$) && ($y == true$) { $z = 5$ } else { $z = 10$ }” (where x and y are type Boolean, and z is type int). But it cannot express the logic of “the only difference between a and b is that a has property c ”. That is because the quantification can only apply to individual entities and not to a group of entities. Plural quantification, e.g. quantifying over “ a and b ” cannot be done in first order logic.

Also, in first order logic, first order functions can only operate on individual entities such as strings, integers, floats, etc. In first order logic, functions cannot operate on functions.

Second order logic deals with quantifying functions and predicates, rather than just quantifying the individual entities. Second order logic can quantify over properties of entities, and to other relations and functions. Quantifiers also apply to entire collections of entities rather than only individual entities; and quantify over the sub-entities of entities (that is, where a separate entity is a property of another entity).

Since second order logic can be applied to the properties of groups of entities, it is second order logic when it is asserted that two entities are equal only if all their attributes have the same properties. In first order logic, it cannot be asserted that “if a is an X , and b is an X , both a and b have a common property”. It also cannot be asserted in first order logic that “ $a == b$ if and only if they share all properties”. Both sentences require second order logic to process.

Go as a language does support both first order and second order logic. In some cases, the support is inherent in the logic of the language itself (e.g. applied by the compiler), and in other cases the program code expresses the logic.

As an example of where the language itself enables second order logic, consider static typing. If it is asserted that: a is a type T , that b is also the same type T , and that c is also the same type T ; therefore, collectively a , b , and c all have the common properties of T . This is obviously true. Go is a statically typed language, and where entities are the same type, then they have common properties of that type.

This static typing which supports second order logic applies limitations to operations which are restricted to entities of given types. For example, while it is possible to assign the value of a variable of type *int* to another variable of type *int*, it is not possible to assign the value of a variable of type *float32* to a variable of type *int32* without a type conversion. The static type system which implicitly supports second order logic within the language comes with logical restrictions upon program code statements.

Go supports the second order logic of quantifying over sub-entities of entities by permitting types to be values of higher types via the use of embedding. So, when a struct of type *A* embeds an anonymous struct of type *B*, this supports second order logic processing. The ability to pass functions to functions as parameters (functional composition) is a second order logic, as a second order function can operate on functions. An expression "*f(g(h()))*" is second order logic because it is applying functions to functions.

Within the program code it is possible to compare the properties of data entities, and the values of those properties. It is possible to embed structs within structs, to embed functions within structs, and to pass functions as parameters to functions, and to return functions as return values from functions. Finally, it is possible to express functions as variables. This type of functionality enables Go to support second order logic.

The language does not support logic above second level logic, that is, the levels collectively known as higher level logic.

Logic: Code Example

```
// First order and second order logic
package main

import (
    "fmt"
    "reflect"
)

func fol() { // All first order logic inside this function
    x := true
    y := true
    var z int
    if x == true && y == true {
        fmt.Println("x and y are both true")
    } else {
        fmt.Println("x and y are NOT both true")
    }
    y = false
    if x == true && y == true {
        fmt.Println("x and y are both true")
    } else {
```

```

        fmt.Println("x and y are NOT both true")
    }
    fmt.Println("z is", z)
    if reflect.TypeOf(x) == reflect.TypeOf(y) {
        fmt.Println("x and y are the same type")
    } else {
        fmt.Println("x and y are NOT the same type")
    }
    if reflect.TypeOf(x) == reflect.TypeOf(z) {
        fmt.Println("x and z are the same type")
    } else {
        fmt.Println("x and z are NOT the same type")
    }
}

func sol1(i int) int { // First order logic
    return i * 10
}

func sol2() int { // First order logic
    return 5
}

type st struct {
    fn func() string
}

func main() {
    fol() // All first order logic inside this function
    fmt.Println(sol1(sol2())) // Second order logic here - function composition
    v := sol2 // Second order logic - assigning a function to a variable
    fmt.Println(v())
    f := st{func() string { return "foo" }} // Second order logic - assigning a function to struct field
    fmt.Println(f.fn())
}

/*
Prints the following:
x and y are both true
x and y are NOT both true
z is 0
x and y are the same type
x and z are NOT the same type
50
5
foo
*/

```


Mapping

Hashing is a method of mapping a collection of data of any size to a collection of data of specified size. Hashing depends upon the use of a hash table. A hash table is an associative array structure that permits the mapping of keys to associated values. The mapping is a representation of ordered pairs of data where none of the first elements of the pair ever appears more than once in the map.

Collisions can occur when performing hashing on a hash table. This is when the key value is calculated from the data value, and two or more data values generate the same key. There are several ways to deal with this. One way is to simply overwrite the existing value. But when multiple values may be associated with a key, commonly the separate chaining approach is used.

The separate chaining approach typically stores the associated values in a linked list of structs, or in a slice containing values. When the hashing calculation determines the key from the value, it checks the hash table, finds the key, and then proceeds down the list or through the slice looking for a match. If there is no match, the value is loaded into a new struct which is added to the linked list of structs, or the value is appended to the slice. If there is a match, then whatever functionality that should be performed when a match is found will be performed.

The Go map data type implements an associative array structure, and the Go specification permits the specified size of the map to be changed at runtime. The key type has a restriction in that it must be a data type that comparable, that is, it must be subject to either of the following Boolean equality operators: “==” or “!=”. Maps are not comparable, so trying `map1 == map2` or `map1 != map2`, these are errors and this will not compile. The value type may be any valid Go data type.

Maps do have a length, which can be determined by the length function, e.g. `len(map)`. While map capacity can be assigned during map initialization, the `cap()` function cannot be performed on a map, e.g. `cap(map)` will result in a compile time error.

To add items to a map, assign a value to a new map key. To update, assign a value to an existing key. To remove items from a map use the `delete(map, key)` function by providing the map name and a key value. The `delete(map, key)` function has no return value. To iterate over an entire map, use the range keyword.

Maps are, under the covers, **pointers** to “`runtime.hmap`” structures. Written in the source code, they do not use the usual “*” pointer designation, but they are in fact pointers. So in the case of “`m1 := map[string]int{“foo”:5, “bar”:8}`”, `m1` is actually a pointer of type `hmap` e.g. “`*hmap`”.

Maps are not concurrency safe. Do not permit two or more goroutines to access the same map without using a mutex or some form of channel communication to coordinate access. See [Goroutines](#), [Mutex \(Mutual Exclusion\)](#), and [Channels](#).

Mapping: Code Example

// Demonstrates basic mapping functionality

package main

import (

 "fmt"

 "math/rand"

 "reflect"

)

func main() {

 // Maps are specified in the form: map [key type] value type

 var m1 map[int]string // Map of int keys to string values, m1 is nil

 m1 = make(map[int]string) // Assign m1 to an initialized map, where m1 contains 0:""

 m2 := make(map[string]int) // Allocates and initializes a map m3, where m3 contains "":0

 m3 := make(map[string]int, 5) // Allocates and initializes map m4 with capacity of 5

 fmt.Println("m1 contains:", m1)

 fmt.Println("m2 contains:", m2)

 fmt.Println("m3 contains:", m3)

 var r = []rune("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ")

 a := make([]rune, 5)

 for i := 0; i < 5; i++ {

 s := func() string {

 for i := range a {

 a[i] = r[rand.Intn(len(r))]

 }

 return string(a)

 }()

 m3[s] = rand.Intn(10000) // Load map m4 with pseudorandom string keys and values

 }

 fmt.Println("m3 contains:", m3)

 m4 := map[string]int{"foo": 5, "bar": 8} // Initialize map m1 with two key-value pairs

 fmt.Println("m4 contains:", m4)

 m4["bob"] = 10 // Add new key-value pair to map

 fmt.Println("m4 contains:", m4)

 m4["bob"] = 12 // Update value associated with key "bob" from 10 to 12

 fmt.Println("m4 contains:", m4)

 i := m4["bob"] // Get the value (12) identified by key, assign to variable

 fmt.Println("Variable i assigned value from map m4 key bob:", i)

```
delete(m4, "foo") // Remove the key-value pair with key "foo"
fmt.Println("deleted foo from map m4 using delete(map,key):", m4)

m4["foo"] = 50 // Add new key-value pair to map
fmt.Println("added new key-value pair to map m4, now contains:", m4)

// Show a different way to delete from a map
m := reflect.ValueOf(m4)
// SetMapIndex() will delete the key from the map if val Value equals zero
m.SetMapIndex(reflect.ValueOf("foo"), reflect.Value{}) // Delete foo from map with reflection
fmt.Println("deleted foo from map m4 via reflection:", m4)

for key, value := range m4 { // Iterate with range
    fmt.Println("Iterating over map m4 with range - Key: ", key, "Value: ", value)
}
fmt.Println("Length of map m4:", len(m4))

_, ok := m4["Foobar"] // Check for key not in the map
if !ok {
    fmt.Println("Key 'Foobar' not found in map m4")
}
}

/*
Prints something like the following:
m1 contains: map[]
m2 contains: map[]
m3 contains: map[]
m3 contains: map[XVlBz:1318 baiCM:8511 AjWwh:1445 Hctcu:6258 xhxKQ:3015]
m4 contains: map[foo:5 bar:8]
m4 contains: map[bob:10 foo:5 bar:8]
m4 contains: map[foo:5 bar:8 bob:12]
Variable i assigned value from map m4 key bob: 12
deleted foo from map m4 using delete(map,key): map[bar:8 bob:12]
added new key-value pair to map m4, now contains: map[foo:50 bar:8 bob:12]
deleted foo from map m4 via reflection: map[bar:8 bob:12]
Iterating over map m4 with range - Key: bar Value: 8
Iterating over map m4 with range - Key: bob Value: 12
Length of map m4: 2
Key 'Foobar' not found in map m4
*/
```

Marshalling and Unmarshalling

The term marshalling is used to refer to the action of transforming the in-memory representation of an entity into a data format used for communicating (transmitting) outside of the program. The word marshalling is derived from the Middle English term “*marshal*” and marshalling linguistically means “to set in proper order” or “set out in an orderly manner”.

The Go language depends upon several packages to implement marshalling and unmarshalling. The base level package is the *encoding* package, which provides interfaces used by several other packages. These interfaces enable binary marshalling and unmarshalling, and UTF-8 textual marshalling and unmarshalling.

Of the several packages that use the encoding package interfaces, two of the more commonly used are *json* and *xml*. But there are others of interest, for example the *csv* package provides the ability to read from and write to files containing comma separated variable format.

When using JSON there are several limitations that must be considered. Firstly, for JSON to be able to marshal values, those values must be exportable, and for a variable to be visible outside of the package that contains it, the first character must be capitalized so that it may be exported. Certain types may not be encoded, such as functions and channels. Pointers will be encoded as the values which they reference, and when encoding map data types, the key types must be strings. JSON marshalling encoding converts the provided data into a `[]byte` slice.

JSON unmarshalling accepts a `[]byte` slice and a pointer to a data structure, and will decode from the `[]byte` slice into that data structure. Assuming the data structure is a *struct* type, the unmarshalling compares the string keys from the `[]byte` slice to the struct field names. When they match, the associated data is loaded into the field; where there is no match, no data is copied into the field. This is useful when only a subset of the marshalled data is needed.

XML marshalling offers a couple of choices, one that does straight marshalling and one that does indented marshalling, where each element is placed on a new line preceded by a provided indentation. XML marshalling will work on structs, arrays, and slices, and does not work on channels, maps, and functions. For structs, the first field should be named `XMLName`, and should be of type `Name`. A `[]byte` slice is returned from a call to either marshal function.

XML unmarshalling accepts a `[]byte` slice and a pointer to a data structure, and decodes from the `[]byte` slice into a struct, string, or a slice. Given a struct, the field names must begin with an uppercase character to be exported so that data can be assigned into the fields. Any missing values or attributes will be unmarshalled as zero values.

For both JSON and XML there are many available functions, and the package documentation must be consulted to understand all the available features and functionality. The following code examples are deliberately simple, but they demonstrate the marshalling and unmarshalling features of the JSON and XML packages.

Marshalling and Unmarshalling: Code Example - JSON

```
// Demonstrate JSON marshalling and unmarshalling
package main

import (
    "encoding/json"
    "fmt"
    "os"
)

func main() {
    type Book struct {
        Title      string
        Author      string
        YearPublished int
    }

    type BookTitle struct {
        Title string
        stuff string
    }

    mybook := Book{
        "A Short Cyclopedia of Go",
        "John Tullis",
        2019,
    }

    fmt.Printf("Initial struct data: %+v\n", mybook)

    // Signature: func Marshal(v interface{}) ([]byte, error)
    bookdata, err1 := json.Marshal(mybook)
    if err1 != nil {
        fmt.Printf("JSON marshalling failed: %s", err1)
        os.Exit(1)
    }
    fmt.Printf("Marshaled data in []byte slice: %s\n", bookdata)

    var getbook Book
    // Signature: func Unmarshal(data []byte, v interface{}) error
    err1 = json.Unmarshal(bookdata, &getbook)
    if err1 != nil {
        fmt.Printf("JSON marshalling failed: %s", err1)
        os.Exit(1)
    }
    fmt.Printf("Unmarshalled into new struct: %+v\n", getbook)

    var booktitle BookTitle
```

```
// Now unmarshal into different struct with only one matching field
err1 = json.Unmarshal(bookdata, &booktitle)
if err1 != nil {
    fmt.Printf("JSON marshalling failed: %s", err1)
    os.Exit(1)
}
fmt.Printf("Unmarshalled into different new struct: %+v\n", booktitle)
}
/*
Prints the following:
Initial struct data: {Title:A Short Cyclopedia of Go Author:John Tullis YearPublished:2019}
Marshaled data in []byte slice: {"Title":"A Short Cyclopedia of Go","Author":"John
Tullis","YearPublished":2019}
Unmarshalled into new struct: {Title:A Short Cyclopedia of Go Author:John Tullis YearPublished:2019}
Unmarshalled into different new struct: {Title:A Short Cyclopedia of Go stuff:}
*/
```

Marshalling and Unmarshalling: Code Example - XML

```
// Demonstrate XML marshalling and unmarshalling
package main

import (
    "encoding/xml"
    "fmt"
)

type Animal struct {
    XMLName xml.Name
    E1      string `xml:"ELEM1"`
    E2      string `xml:"ELEM2"`
    E3      string `xml:"ELEM3"`
}

func main() {
    // Signature: func MarshalIndent(v interface{}, prefix, indent string) ([]byte, error)
    fmt.Println("Marshalling dog data with indentation")
    buf, err := xml.MarshalIndent(Animal{
        XMLName: xml.Name{Local: "Dog"},
        E1:      "bark",
        E2:      "bark",
        E3:      "more barks",
    }, "", " ")
    if err != nil {
        fmt.Println("error:", err)
    } else {
        fmt.Printf("%s\n", buf)
    }
}
```

```

var newdog Animal
fmt.Println("New struct newdog contains:", newdog)
fmt.Println("Unmarshalling dog data")
// Signature: func Unmarshal(data []byte, v interface{}) error
err = xml.Unmarshal(buf, &newdog)
if err != nil {
    fmt.Println("error:", err)
} else {
    //fmt.Printf("%s\n", buf)
    fmt.Println(newdog)
}

// Do it again
fmt.Println("Marshalling cat data without indentation")
// Signature: func Marshal(v interface{}) ([]byte, error)
buf, err = xml.Marshal(Animal{
    XMLName: xml.Name{Local: "Cat"},
    E1:      "meow",
    E2:      "hiss",
})
if err != nil {
    fmt.Println("error:", err)
} else {
    fmt.Printf("%s\n", buf)
}

var newcat Animal
fmt.Println("New struct newcat contains:", newcat)
fmt.Println("Unmarshalling cat data")
err = xml.Unmarshal(buf, &newcat)
if err != nil {
    fmt.Println("error:", err)
} else {
    //fmt.Printf("%s\n", buf)
    fmt.Println(newcat)
}
}
/*
Prints the following:
Marshalling dog data with indentation
<Dog>
  <ELEM1>bark</ELEM1>
  <ELEM2>bark</ELEM2>
  <ELEM3>more barks</ELEM3>
</Dog>
New struct newdog contains: {{ } }
Unmarshalling dog data
{{ Dog} bark bark more barks}

```

Marshalling cat data

```
<Cat><ELEM1>meow</ELEM1><ELEM2>hiss</ELEM2><ELEM3></ELEM3></Cat>
```

New struct newcat contains: {{ } }

Unmarshalling cat data

```
{{ Cat} meow hiss }
```

```
*/
```


Method Set

A method set must be associated with a type, and there are two kinds of method sets. First, the method set of a data type is the set of all methods declared with the receiver of that type. Second, the method set of an interface type is its interface. Each method in a method set must have a unique (non-blank) name. A method set may contain only one method. For interfaces, see [Interface](#).

Method receivers can either be specified as a type, or a pointer to a type. If the method is not intending to mutate its receiver, should the method be defined as receiving a type pointer?

There are five reasons why the answer should be yes:

1. If a method intends to mutate its receiver, its receiver must be a receiver type pointer.
2. The method sets for T and *T are different. The method set for a receiver T is only the methods that also receive T; but the method set for a receiver *T contains both the methods that receive *T **and** the methods that receive T.
3. The Go FAQ asserts that if one method for a receiver T must have a receiver type T pointer, then for consistency all the methods for T should receive a type pointer.
4. Passing a pointer to an instantiated type into a method has lower overhead than passing the instantiated type value itself, this is especially true if the type instance is large.
5. If the receiver is a struct containing a mutual exclusion lock, then the receiver must be a pointer to the instantiated type to avoid unsafe or unexpected behavior.

In practice, preferentially using the *T receiver provides multiple benefits. See the following code example for a data type with two methods, the pair of methods compose the method set. There is also an interface that matches one of the methods.

Method Set: Code Example

```
// Show the method sets for struct and interface types
package main

import (
    "fmt"
)

type Displayer interface { // By convention single method interface "-er"
    display()
}

type Book struct { // Book declared to be type struct
    title string
    author string
    pubdate int
}
```

```
func (b *Book) load(t, a string, p int) { // Book method to load struct
    b.title = t
    b.author = a
    b.pubdate = p
}

func (b *Book) display() { // Book method to display contents
    fmt.Printf("Title: %s, Author: %s, Publication Date: %d\n", b.title, b.author, b.pubdate)
}

func main() {
    var b Book // Declare instance of Book
    var d Displayer // Declare variable of interface type Displayer
    b.load("The Fast Rabbits", "Jack Rabbit", 2019) // Assign values to b via method
    b.display() // Show book b values via method
    d = &b // Method has pointer receiver, interface does not
    d.display() // Show book b values via interface
}

/*
Prints the following:
Title: The Fast Rabbits, Author: Jack Rabbit, Publication Date: 2017
Title: The Fast Rabbits, Author: Jack Rabbit, Publication Date: 2017
*/
```

Multiplexing

Multiplex is derived from Latin “*multus*” meaning much or many, and “*plex*” (meaning parts) which came from “*plectere*” meaning to plait or braid (those parts). In programming it refers to interleaving several activities or combining several streams into one stream.

Go implements this concept via the select statement. The general case is where one goroutine (which may be the main goroutine) is receiving input from several other goroutines or sending output to several other goroutines. Often the select statement is enclosed within a loop, this is done when the goal is to process multiple channels instead of only handling the first channel that provides input to the select (or the first of which is selected to which a message is sent).

The select will evaluate all the channels. If no channel operation may be executed, then the default case (if provided) will be processed. If no default case is provided, the select will block until one or more channel operations are possible. When one or more channel operations may be performed, then by a pseudo-random selection, one will be chosen.

For receive operations the select case statement may use one argument or two arguments. If using one argument, that argument is the variable into which the incoming message is assigned. Here a default case must be provided, because every closed channel returns a nil, and if all channels are nil, without a default the select will block forever.

With two arguments provided for a channel receive operation the purpose of the second argument is to determine if a channel is closed or not. In this case the select will not block but will return a nil. Provide logic to handle the situation when a nil is returned (channel is closed).

There are several possible use cases. One might be where the receiver will listen for multiple senders, but whichever sender transmits to the receiver first is the one that gets processed and the others are ignored.

Another use case is where the receiver will listen for multiple senders and will handle all of them. In this situation the select statement must be placed inside a loop. Assume each transmitting goroutine will run until completion, and once each goroutine is finished, it will close the transmission channel. The receiving goroutine will listen selectively on all channels. Whenever a channel closes, the receiving goroutine knows that there will be no more incoming data coming from that channel, so sets a flag to indicate that channel is done.

Once all channels are marked as closed, the receiving goroutine can finalize processing and exit. See the following code as an example of the main goroutine working with two other goroutines using channels. Note the main goroutine passes the channels to the other two goroutines, this practice is recommended rather than using global variable channels.

Multiplexing: Code Example

```
// Demonstrating the concept of multiplexing
```

```
package main

import (
    "fmt"
    "time"
)

var chn1 chan string
var chn2 chan string
var ch1closed bool
var ch2closed bool

func init() { // Demonstrating use of init() function - notice not called in main
    chn1 = make(chan string) // Create an unbuffered (synchronous) channel
    chn2 = make(chan string) // Create 2nd unbuffered (synchronous) channel
    ch1closed = false        // Initialize channel 1 status variable
    ch2closed = false        // Initialize channel 2 status variable
}

func mygo1(ch1 chan<- string) { // Function with writable only channel param
    for i := 0; i < 3; i++ {
        ch1 <- "foo"
    } // Send "foo", block until received, loop
    close(ch1) // Close channel 1 and exit goroutine
}

func mygo2(ch2 chan<- string) { // Function with writable only channel param
    for i := 0; i < 5; i++ {
        ch2 <- "bar"
    } // Send "bar", block until received, loop
    close(ch2) // Close channel 2 and exit goroutine
}

func main() {
    go mygo1(chn1) // Start goroutine, pass channel 1 as param
    go mygo2(chn2) // Start goroutine, pass channel 2 as param

    for i := 0; i < 30; i++ { // Loop limited number of times
        fmt.Println("Loop iteration: ", i+1)
        time.Sleep(time.Millisecond * 1000) // Pause to give goroutines time to run
        select {
            // Selectively listen to both channels
            case msg1, ok := <-chn1: // Listen for transmission from channel 1
                if ok {
                    fmt.Println("Received", msg1) // Channel is open, message received
                } else {
                    if ch1closed != true { // If ch1closed == true then skip this
                        fmt.Println("Setting ch1closed = true")
                        ch1closed = true // Set flag to indicate channel 1 is now closed
                    }
                }
            }
        }
    }
}
```

```
case msg2, ok := <-chn2: // Listen for transmission from channel 2
    if ok {
        fmt.Println("Received", msg2) // Channel is open, message received
    } else {
        if ch2closed != true { // If ch2closed == true then skip this
            fmt.Println("Setting ch2closed = true")
            ch2closed = true // Set flag to indicate channel 2 is now closed
        }
    }
} // End select
if (ch1closed == true) && (ch2closed == true) {
    fmt.Println("Exiting loop")
    break // Break if out of loop both channels closed
}
} // End of for loop
fmt.Println("Program exits")
}
/*
Prints something like:
Loop iteration: 1
Received bar
Loop iteration: 2
Received foo
Loop iteration: 3
Received bar
Loop iteration: 4
Received foo
Loop iteration: 5
Received foo
Loop iteration: 6
Received bar
Loop iteration: 7
Received bar
Loop iteration: 8
Setting ch1closed = true
Loop iteration: 9
Received bar
Loop iteration: 10
Loop iteration: 11
Setting ch2closed = true
Exiting loop
Program exits
*/
```

Mutex (Mutual Exclusion)

The concept of mutual exclusion is that it may be necessary to prevent two or more events happening simultaneously. Usually it is acceptable for multiple data read actions to occur together, but not actions that update data. When two or more processing threads have access to the same data with the intent of updating the data, a race condition may occur.

A race condition occurs when sequences of actions by individual processing threads on a data item results in unexpected or indeterminate state change. Two common problem examples are: “read then update”, and “check then act”.

1. Suppose there is code to read a data value, increment it, and store back the incremented value. But when two threads are both running, the result might be: read, read, increment, increment, store, store. The stored value should be incremented twice, but it is only incremented once because both store actions overwrite the data value with the same changed value. So, if the data value is 1, both threads read in 1, both threads increment 1 to 2, and then both threads store value 2. But the desired behavior is read increment store, read increment store, e.g. 1 incremented to 2, then 2 incremented to 3.
2. For check then act, suppose there is an inventory system that must be checked before permitting an order to allocate the item. In the case where there is 1 item remaining in inventory, for two order threads the desired result is that one order gets allocated the item and decrements the inventory to 0, and the other order is informed the item is out of stock. But in the race condition: check, check, allocate, allocate, decrement, decrement both orders get allocated the item and inventory is decremented to -1 (or the 2nd decrement throws an error). The problem of course is only 1 order can be filled.

The solution to prevent the race condition is to enforce mutual exclusion. To do this properly means only one thread of control may access a critical area at a time, and deadlocks must be avoided. Go enforces the first constraint via the use of mutual exclusion locks, or mutex. This is provided by the sync package. This package provides a mutex type and two methods:

```
type Mutex struct { // contains unexported fields }
func (m *Mutex) Lock() { // locks the struct instance m }
func (m *Mutex) Unlock() { // unlocks the struct instance m }
```

The way this works in practice is that if there are two or more goroutines, the first one to lock the mutex gets access to the critical area, and the other goroutines will block until the first one unlocks the mutex. This approach prevents the race condition. This approach can also be done using a synchronous channel, but in good coding practice channels should be used for communications between goroutines, while mutexes should be used to prevent race conditions.

To prevent deadlock on an exclusive access resource, see [Deadlock](#).

Mutex: Code Example

//Illustrating usage of mutex for inventory access

```
package main

import (
    "fmt"
    "sync"
    "time"
)

var (
    mutex sync.Mutex
    store int
)

func init() {
    store = 0
}

func loadInventory(quantity int, done chan bool) {
    mutex.Lock()
    fmt.Printf("Incrementing inventory containing %d by %d\n", store, quantity)
    store += quantity
    mutex.Unlock()
    done <- true
}

func decrementInventory(quantity int, done chan bool) {
    mutex.Lock()
    if store <= 0 { //Check otherwise decrementInventory might run first and result in a negative 3
        mutex.Unlock()
        time.Sleep(1000 * time.Millisecond)
        decrementInventory(3, done)
    } else {
        fmt.Printf("Withdrawing %d from inventory containing store: %d\n", quantity, store)
        store -= quantity
        mutex.Unlock()
        done <- true
    }
}

func main() {
    done := make(chan bool)
    go loadInventory(10, done)
    go decrementInventory(3, done)
    <-done
    <-done
    fmt.Printf("Current inventory %d\n", store)
```

}

/*

Will produce the following:

Incrementing inventory containing 0 by 10

Withdrawing 3 from inventory containing store: 10

Current inventory 7

*/

Package

A package is a container that contains either a single part, or a group of related parts; the term comes from the Dutch word “*pakage*” which means baggage. Within Go code functionality a package contains a collection of interrelated functionality.

Packages are organized by directory sub-folders under the Go “/src” folder. All files within the same sub-folder should belong to the same package. For any package to access code from other packages, those packages must be imported. Packages have a path and a name, by convention, the package name should be the same as the last element of the path. The Go compiler searches for packages via the GOROOT and GOPATH environmental variables.

Within a package, all fields are visible. But if the package is imported by another package, only the fields beginning with capital letters are visible. Capital letter means Unicode uppercase letters which are in category “Lu”. There are currently 1702 characters in this category.

The following example shows a number of interesting things:

1. Three packages, main, demo, and fmt; package main imports packages fmt and demo
2. All files in a package must have their package name as the first line of the file
3. Package demo declares a struct type (“*myval*”) lowercase, so it is not exported
4. However the struct has a capitalized field which may be visible outside of demo
5. The function “*Newval()*” in package demo is exported because it is capitalized
6. Therefore function “*main()*” can call “*demo.Newval()*” and assign results to “*i*”
7. When variable “*i*” is assigned the address of struct *myval*, it can see variable “*Val*”

package main	// Package main is in its own file in its own directory
import (
“fmt”	// Compiler finds this standard package from GOROOT
“demo”	// Compiler finds this custom package from GOPATH
)	
main{	
i := demo.Newval(100)	// Variable i is declared and assigned pointer to myval struct
fmt.Println(i.Val)	// Prints 100 since Newval() assigned 100 to struct field Val
}	
package demo	// Package demo is in its own file in a separate directory
type myval struct {	// Type myval is not exported because myval not capitalized
Val int	// But capitalized field is visible externally of package demo
}	
func Newval(j int) *myval {	// Function name capitalized so visible outside of package demo
return &myval{j}	// Declare struct, assign 100 to Val field, return struct address
}	

Methods can be attached to any type; however, they can only be attached to types in the same package. This enforces encapsulation. See [Encapsulation](#) and [Method Set](#).

Parallelism

Suppose a program emulates a person walking around a lake. The program has functions for talking, walking, and tying shoes. All three of these functions are concurrent. In a single core system, only one of these functions may be executed at a time. Context switching may allow switching between these functions rapidly, but only one may be done at a time.

Parallelism requires access to multiple cores and concurrency. Unlike concurrency alone, with parallelism program code may be executed in multiple cores simultaneously. For reasons to use concurrency other than enabling parallelism, see [Concurrency](#).

In a multicore system that permits parallelism, both walking and talking can occur in parallel. However, the shoe tying function will block until a shoe comes untied. When a shoe comes untied, shoe tying unblocks but walking blocks. Walking cannot resume until the shoe tying function completes. So, in this thought experiment, parallelism permits walking and talking in parallel, or talking and tying in parallel, but not walking and tying in parallel. All three functions may be concurrent, but they are not all able to take advantage of parallelism simultaneously.

By default, in Go the number of logical processors allocated to a program is set via the GOMAXPROGS environmental variable. This variable may be set to change core utilization by importing the “runtime” package and setting the variable. However ever since Go version 1.5 this is set to the maximum number of available cores. See [Environmental Variables](#).

No code example is provided for parallelism as it is not possible to demonstrate this with the Go playground, and even on a multicore system demonstrating the actual usage of multiple cores simultaneously requires observational monitoring outside of the code itself. Go supports both concurrency, and parallelism.

Patterns

A pattern may be considered as a solution to a general problem within a given context. It may be defined as a model to be imitated or a guide to be followed, the word coming from the Latin “*patrōnus*”. Patterns arise when it is realized that specific solutions to a set of concrete problems that exist within a common context can be abstracted. Patterns are abstractions.

For example, in the physical world a problem might be replacing a flat tire on a vehicle. So, the problem is known (flat tire), the context is known (vehicle might be car, truck, tractor, etc.), and the solution involves a sequence of steps using tools (jack up the vehicle, unbolt wheel from axle, remove wheel with flat tire, put new wheel with full tire onto axle, bolt wheel to axle, jack down the vehicle). The abstracted pattern is the “Replace Flat Tire” pattern.

The pattern is general and conceptual, while the specifics of a given concrete problem will be more varied. It might be raining, the vehicle is on a slope and not a flat surface, there is no replacement wheel in the vehicle, and one must be acquired, and so on. The pattern will not provide all the detailed information required to resolve a concrete situation.

Within computer science there are two separate but overlapping areas of patterns: computer architecture patterns and software design patterns. While in most cases it is obvious into which area a given pattern exists, sometimes a pattern may be applicable to both areas.

Architectural patterns are generally reusable solutions to common problems within the domain of computing architecture. Architecture includes hardware (processing and network) as well as virtual machines, containers, and infrastructure software servers (web, application, database). Computing architecture is then the entire environment (including tooling) that supports software programs. Examples of architectural patterns include “Data Warehouse”, “Enterprise Service Bus”, “Extract, Transform, Load”, and “Model View Controller”.

Software design patterns are generally reusable solutions to common problems within the domain of software design. This includes all the software (excluding infrastructure software) that is used to execute solutions to computing problems.

An observation regarding many extant software design patterns is that they appear to have been created to overcome limitations within object-oriented languages. Limitations exist in the ways that the various languages implement support for the concept of objects, and how the objects relate to solving real world problems. These kinds of software design patterns are not applicable to Go as they are simply not needed.

Patterns applicable to Go leverage features of the language to solve problems, rather than being patterns that exist to overcome deficiencies in the language. The implementation of these patterns leverages how the language supports concepts such as composition, concurrency, embedding, and interfaces. The following are typical patterns used with Go.

- **Adapter:** permits entities of different types to communicate when their interfaces are incompatible. The way the adapter pattern works is that calls can be made to the adapter via its interface (the target), and the adapter then calls the interface of the adaptee. Go does not handle this the way a language that has classes and inheritance would handle it, but instead uses embedding, where one struct (the adapter) will embed another struct inside it as an anonymous field. As such, the embedding (outer) struct has access to the methods of the embedded (inner) struct. So, when a call is made to an interface that maps to a method of the adapter type, that method then handles the call to a method of the embedded adaptee type. See [Embedding](#).
- **Composite:** provides access to a tree structured collection of entities, the interface of the collection is the same as the interface to any individual entity. Whether the entities are parent nodes or child (leaf) nodes in the tree, they have the same interface. In Go this may be implemented such that the entities are structs, and the interface will provide basic actions like add, remove, and display. See [Interface](#).
- **Decorator:** permits extending functionality of an existing entity without altering its structure. Similarly, to the way Go implements the adapter pattern, the approach is to have one struct embed another struct. The embedding struct provides additional methods beyond the methods of the embedded struct. Calling the interface of the embedding struct provides access to both the additional methods of the embedding struct and the original methods of the embedded struct.
- **Entity Constructor:** a function that returns an entity. Unlike object-oriented programming where a call to a constructor returns an object (which contains data entities, control flow logic, and methods), in Go an entity constructor returns only a data entity. A constructor can be useful in providing the following kinds of data entities:
 - A condition variable that enables communication between goroutines
 - A mutual exclusion lock that enables coordination of access between goroutines
 - A channel that enables communication between goroutines
 - A specific struct that will be communicated between goroutinesIn each of these examples, the constructor is used to create data entities that are used by goroutines. See [Channels](#), [Condition Variable](#), and [Mutex \(Mutual Exclusion\)](#).
- **Iterator:** provides the capability to traverse a container and access the contents of the container without having to know the data structures within the container. Go easily implements this within the language. For example, suppose there is an array containing a diverse collection of types, each of which has an identically named method; and an interface is provided which all these types satisfy. Then simply performing a range command on the array as a *for* loop condition, and calling the interface within the loop, will permit iterating across the container and accessing the contents without having to know the type of any entity within the container. See [Polymorphism](#).
- **Mediator:** entities communicate with each other through a common mediator. In Go the mediator is an interface. See [Interface](#).
- **Messaging request-response:** a two-way communication pattern between goroutines. This may be implemented with either synchronous or asynchronous channels. See [Channels](#).

- Messaging publish-subscribe: a one-way communication pattern between goroutines. This may be implemented with either synchronous or asynchronous channels.
- Monitor: allows goroutines to have both mutual exclusion and block waiting, and the ability to signal other goroutines. The language enables this with goroutines, channels, and the sync package which provides synchronization primitives such as mutual exclusion (mutex) via locks, and the condition variable (which permits broadcasting to all goroutines waiting for an event). See [Condition Variable](#) and [Mutex \(Mutual Exclusion\)](#).
- Observer: a goroutine (the subject) has dependent goroutines (the observers) who wish to be notified when a change of state (an event) occurs. This notification may be provided either with channels or a condition variable. The key difference between the Observer pattern and the Messaging publish-subscribe pattern is that the former is communicating information about an event, while the latter is communicating data which is to be used for some purpose. See [Goroutines](#).
- Pipes and Filters: also called the pipeline pattern. A pipeline is a pattern that refers to a flow of data through a sequence of tasks that operate on the data. In the Go context the pipeline pattern depends upon goroutines and channels. See [Concurrency](#), [Channels](#), [Goroutines](#), and [Parallelism](#).
- Proactor: uses proactive event dispatching for goroutines to execute asynchronous operations. This is a client-server model where the server must respond to concurrent client requests, and where a separate dedicated handler is spawned to handle each connected client. See [Concurrency: Code Example](#).
- Producer Consumer: implemented with goroutines and channels. May be a fan in pattern which is many producers and one consumer, or a fan out pattern which is one producer and many consumers, or a many to many pattern. See [Goroutines](#) and [Channels](#).
- Scheduler: a pattern to optimize how work is done by multiple goroutines assigned to various resources. See [Goroutines](#), [Multiplexing](#), and [Periodicity](#).
- Singleton: this is where a single entity exists, but where that entity is accessed by multiple concurrent goroutines. Access is controlled via the use of condition variables, mutual exclusion, and wait groups. See [Condition Variable](#) and [Mutex \(Mutual Exclusion\)](#).

There are other patterns of course: Fan-In which directs multiple tasks to a single task processor, Fan-Out where a single task assigner hands tasks to multiple workers, Circuit Breaker which halts task assignments when load is too high to handle (or backlog is too large), Facade where one API shields multiple other APIs, and so on. However, upon investigation it becomes clear that Go has far fewer design patterns than most object oriented languages.

Periodicity

Periodicity means something that recurs at regular intervals, with the attribute of being periodic. The term comes from the French word “*périodicité*”, derived from Latin “*periodicus*” and Greek “*periodikós*”. In the context of computer programming it relates to activity that is scheduled and recurring. Commonly this is used for polling, and for heartbeat keepalive functionality.

In the situation of polling or keepalive signals, this maps to a request-response scenario, where a goroutine or application regularly checks for the existence of something, or the status (state) of something. For heartbeat transmissions this maps to a publish-subscribe scenario, where a goroutine or application periodically transmits a notification that it is still alive and functional.

Polling or keepalive are active approaches, which differs from the wait state approach that might use channels or condition variables to wait until new data arrives or a state change occurs. It is generally used when an application or goroutine has other activities to perform and so just checks periodically on whether new data has arrived, or a state has changed (keepalive is checking on the state or status of something else). This is also the case for sending a heartbeat, where an application or goroutine is busy but periodically transmits to something else to communicate it is still functioning.

To support this type of functionality Go has several data types and functions in the *time* package. Most useful here are *Timers* and *Tickers*.

The ticker feature consists of a data type *Ticker* which is a struct containing a channel. The *NewTicker(duration)* function returns a pointer to a struct of type *Ticker*. Once created, every time period equal to the duration passed to the ticker creation results in a time duration message written to the channel. The ticker can be stopped by a call to *ticker.Stop()*. Once it is stopped it cannot be restarted. While the stop function prevents any more time ticks from being sent to the ticker channel, it does not close the channel. Therefore, if a range statement is being used on the ticker channel, it will hang on the channel read once the channel is empty of ticks.

The timer feature differs from the ticker in that it executes only once. However, it can emulate a ticker if it is placed into a loop. Like the ticker, the data type *Timer* is a struct containing a channel. The function *NewTimer(duration)* returns a pointer to a struct of type *Timer*, and after a time period equal to the duration passed to the timer creation the current time will be sent to the *Timer* channel. There is a *timer.Stop()* function that will attempt to cancel the timer, and there is a *timer.Reset(duration)* function that will attempt to reset the timer to a new duration. Finally, there is the *timer.AfterFunc(duration, f())* function, which after the provided duration passes, will execute the function you provide as a goroutine.

In both cases periodicity is done by waiting on a channel for a time period equal to duration. If using the *range* statement a mechanism must be provided to prevent permanently hanging on a channel read when a *ticker.Stop()* or *timer.Stop()* is issued. One way to do that is to instead

execute the channel check via a *select* statement within a *for* loop, with another channel provided to indicate the timer or ticker is no longer active.

In general, a ticker is better used for periodicity, but in some cases contriving to use a timer within a loop may be better. This is because timer has the two additional functions, with *Reset()* being able to change the time duration interval, and *AfterFunc()* permitting the specification of a given function to be executed.

The following code example shows several features of timers and tickers. A ticker is used for three goroutines: one anonymous function, function *myf1()*, and function *myf2()*. Both the anonymous function and function *myf1()* use a counter, whereas function *myf2()* uses *select* and a second *done* channel, to know when they are done processing. If a ticker is stopped before a counter is fully decremented, or when not using *select* with a second channel to know when processing is done, then the *range* statement on the ticker will hang. This is because when applied to a channel, *range* only returns if it has a value, or when the channel is closed. But the *ticker.Stop()* call does not close the ticker channel, it just stops the ticker.

In the following example code, the anonymous goroutine counter counts down completely before the ticker is stopped, so it completes. The *myf2()* goroutine detects the *done* channel is closed, so it completes. But *myf1()* keeps looping because its counter is not fully decremented, so when the ticker is stopped, it hangs on the *range* statement.

Periodicity: Code Example

```
//Illustrating periodicity
package main

import (
    "fmt"
    "time"
)

func myf1(ticker *time.Ticker) {
    counter := 5 // Needs a counter to exit
    for _ = range ticker.C { // Will hang on channel read if ticker stopped
        fmt.Println("myf1() goroutine tick")
        counter-- // Decrement the counter
        if counter == 0 { // Will loop 5 times or until ticker stopped
            fmt.Println("myf1() goroutine done with ticker")
            return
        }
    }
}

func myf2(ticker *time.Ticker, done chan bool) {
    for { // No counter required, use done channel to enable exit
        select {
```



```

        case <-ticker.C:
            fmt.Println("myf2() goroutine tick")
        case <-done:
            fmt.Println("myf2() goroutine done") // 2nd channel indicates time to exit
            return
    }
}

```

```

func main() {
    ticker := time.NewTicker(time.Millisecond * 500)
    done := make(chan bool, 1)
    go myf1(ticker)
    go myf2(ticker, done)
    go func() {
        counter := 3 // Needs a counter to know when to exit
        for _ = range ticker.C { // Hang on channel read after the stop
            fmt.Println("Anonymous goroutine tick")
            counter-- // Decrement the counter
            if counter == 0 { // Will loop 3 times or until ticker stopped
                fmt.Println("Anonymous goroutine done with ticker")
                return
            }
        }
    }()
    timer1 := time.NewTimer(time.Millisecond * 3500) // Set timer to let goroutines run
    <-timer1.C // Ignore result, don't care
    fmt.Println("Timer 1 expired")
    ticker.Stop() // Stop the ticker, this does not close the channel!
    fmt.Println("Ticker stopped - this will hang myf1()")
    time.Sleep(time.Millisecond * 2000) // Let goroutines run using Sleep not timer for fun
    close(done) // Closing this channel tells myf2() to exit
    if timer1.Stop() == true { // Safety measure - drain if not done before reset
        fmt.Println("Won't see this, timer1 has run down already")
        <-timer1.C // But if it hadn't, and Stop() stops it, drain here
    }
    timer1.Reset(time.Millisecond * 2000) // Reset timer to let goroutines run
    <-timer1.C // Ignore result, don't care
    fmt.Println("Exiting program, notice goroutine myf1() is hanging")
}

```

```

/*
Should print something like:
Anonymous goroutine tick
myf1() goroutine tick
myf2() goroutine tick
Anonymous goroutine tick
myf1() goroutine tick
myf2() goroutine tick

```

```
Timer 1 expired
Ticker stopped - this will hang myf1()
Anonymous goroutine tick
Anonymous goroutine done with ticker
myf2() goroutine done
Exiting program, notice goroutine myf1() is hanging
*/
```

Polymorphism

Go implements polymorphism by presenting the same interface to different data types that will make use of the interface. The word is derived from Greek: “*poly*” meaning many, and “*morph*” meaning shape (English has changed morph to mean “change shape”). Go implements polymorphism via the use of interfaces, this approach is known as ad hoc polymorphism. This type of polymorphism matches the definition used by Bjarne Stroustrup (creator of C++) which is “providing a single interface to entities of different types”. Go does not support parametric polymorphism (generic functions and generic types). See [Interface](#).

Two or more data types may generate a different response for calls to a given interface method. For example, suppose there was an array of media types (books, magazines, newspapers, periodicals, etc.). Cycling through the array, accessing the common interface would produce different results based on the media type encountered in the current array cell.

Furthermore, interfaces may embed other interfaces, this behavior is another aspect of interface polymorphism. For example:

- Suppose an interface is declared with the name “*mediaTyper*”
- Suppose the interface has the method “*mediaType()*”
- Suppose an array was defined to contain items of “*mediaTyper*” (the interface type)
- Suppose multiple aggregate data types are assigned into the array
- Suppose each type in the array has implemented a “*mediaType()*” method, therefore each data type in the array has satisfied the interface
- Suppose the interface is called for each item in the array
- Then each media type could respond differently to the call (e.g. “I’m a book”, “I’m a magazine”, “I’m a newspaper”, etc.)
- Suppose further that another interface embeds the first interface (e.g. *embeddedMediaTyper* embeds *mediaTyper*).
- Suppose an array was defined to contain items of *embeddedMediaTyper*. Then by calling the embedding interface, the method of the embedded interface is accessible.

The following code will first produce: “I am a Media Type, and I’m a Book”, then “I am a Media Type, and I’m a Magazine”, and finally “I am a Media Type, and I’m a Newspaper”; and then when calling the embedding interface: “I am an embedded Media Type, and I’m a Book”, then “I am an embedded Media Type, and I’m a Magazine”, and finally “I am an embedded Media Type, and I’m a Newspaper”

Polymorphism: Code Example

```
//Illustrating ad hoc polymorphism
package main

import (
    "fmt"
)
```

```

type mediaTyper interface {
    mediaType() string
}

type embeddedMediaTyper interface {
    mediaTyper
} // Interface embedding another interface

type Book struct{}
type Magazine struct{}
type Newspaper struct{}

func (b Book) mediaType() string      { return "I'm a Book." }
func (m Magazine) mediaType() string { return "I'm a Magazine." }
func (n Newspaper) mediaType() string { return "I'm a Newspaper." }

func main() {
    b := new(Book)
    m := new(Magazine)
    n := new(Newspaper)
    mtArr := [...]mediaTyper{b, m, n}
    emtArr := [...]embeddedMediaTyper{b, m, n}
    fmt.Println("Method calls, not polymorphic")
    for i := range mtArr {
        fmt.Printf("I am a Media Type, and %s\n", mtArr[i].mediaType())
    }
    fmt.Println("Ad hoc polymorphism demonstrated by interface calls")
    for i := range emtArr {
        fmt.Printf("I am an embedded Media Type, and %s\n", emtArr[i].mediaType())
    }
}

/*
Prints the following output:
Method calls, not polymorphic
I am a Media Type, and I'm a Book.
I am a Media Type, and I'm a Magazine.
I am a Media Type, and I'm a Newspaper.
Ad hoc polymorphism demonstrated by interface calls
I am an embedded Media Type, and I'm a Book.
I am an embedded Media Type, and I'm a Magazine.
I am an embedded Media Type, and I'm a Newspaper.
*/

```

Race Condition

A race condition in software program code occurs when concurrent code makes assumptions regarding the timing of external events, and the events occur in an unexpected order. One situation is when multiple routines check a value state, and then perform an action, and then change the value state. Another situation is when multiple routines copy a common variable, increment the copy, and write the copy back to the common variable.

Without proper coordination, the result is that an action may be performed too many times, or data becomes overwritten, and the results are unexpected or incorrect. In Go this may occur when goroutines access a common data structure without ensuring proper locking before and after accessing that data structure.

As the playground is a deterministic environment, race condition results will not occur in the playground even if the code permits race conditions. For this reason, the example code must be compiled and executed in a local environment. In the playground the result is always $x = 1000$, because the playground is preventing the race condition.

Race Condition: Code Example

```
// Race condition demonstrated
// Copy code into a text file, name it race.go; compile with "go build race.go"; execute with ".\race.exe"
package main
```

```
import (
    "fmt"
    "sync"
)

func main() {
    var w sync.WaitGroup
    x := 0
    for i := 0; i < 1000; i++ { // Launch 1000 goroutines
        w.Add(1) // Alternatively could do w.Add(1000) before the loop
        go func() {
            defer w.Done()
            x = x + 1
        }()
    }
    w.Wait()
    fmt.Println("Final value of x", x)
}
/*
```

Will produce varying output such as the following when repeatedly executed:

Final value of x 974

Final value of x 986

Final value of x 959

Etcetera

If this code is executed in the Go playground, result will always be:

Final value of x 1000

*/

Recursion and Memoization.

Recursion occurs when the steps of a procedural algorithm involve the procedure calling itself. Recursion must have a base case where the procedure does not call itself, as recursion should not recur infinitely. The word comes from the Latin word “*recursi*” meaning the act of running back or returning.

Mathematics depends upon the principle of recursion in several situations. For example, the set of natural numbers can be defined recursively as follows: zero is in the set of natural numbers, if n is in the set of natural numbers then $n + 1$ is itself a natural number. By this axiom all of the natural numbers are therefore specified e.g. 0, 1, 2, 3, ... out to infinity.

There are many common code examples using recursion, such as calculating prime numbers, calculating the Fibonacci numbers, the tower of Hanoi, and so on. These are useful for teaching the concept but are not very useful in the world of reality. Recursion is more expensive in terms of computing resources than iteration, but the code can be much simpler.

In general, the most common applicability for using programmatic recursion is when the problem space consists of trees or lists. This is because trees contain trees and lists contain lists. Thus, a common algorithm specified in a function that processes the tree or the list can call itself recursively to process through the tree or list.

In the real world there are problems where recursion is useful. An example of this is the concept of a bill of materials, or BOM. In manufacturing an item is often composed of many assemblies, which are themselves composed of sub-assemblies, which ultimately are composed of parts. These parts may be purchased from suppliers or manufactured locally from materials. The parts are composed of materials, and the materials themselves may be specified as purchased or manufactured on site. An example of a material might be insulated copper wire. Note that a BOM is essentially a tree that contains lists at the leaf nodes, but the BOM data is usually stored in tabular (list) format in a database.

Now the BOM may be recursively analyzed to determine time and to determine cost for manufacturing any item. The code for such a recursion will be much simpler to create and to understand than iterative code.

Memoization is derived from the late Middle English word “*memorandum*”, which refers to a short note to be remembered, and further back from the Latin “*memorandus*” meaning something to be noted. In modern American English this is shortened to *memo*. Therefore, in programming memoization means that the results of a function call are to be remembered.

Memoization is useful for function calls that are expensive to call and where the function always returns the same value, given the same input. The concept of memoization is that the results of expensive operations are stored in a cache, and the cache can be checked before calling the expensive function. The cache check takes very little time and is an inexpensive operation.

If the function has previously calculated the response for a given input, the result will be found in the cache; if it is not found, the function is called which will perform the operation. Either the function will update the cache before returning the value, or the caller may do so after receiving the result. A map containing key value pairs is a good data structure for such a cache, where the key maps to a provided input parameter, and the value is the calculated response.

The following code example shows three recursive functions, performing calculations for prime numbers, for the Fibonacci sequence, and for determining factorials. The first two functions called are purely recursive, but the factorial function enables memoization via storing the calculated results in a map cache. It is good practice to pass the map as a parameter.

The caller can check the map cache first before calling the function, if the value is found in the map there is no need to call the factorial calculation function. The function is called in two separate loops. In the first pass the function is called every time, it updates the map, and returns the values. In the second pass some of the values are found in the cache and so the function is not called in those cases.

Recursion: Code Example

```
// Demonstrate recursion and memoization for primes, Fibonacci sequence, and factorials
package main
```

```
import (
    "fmt"
)

func fact(n int, m map[int]int) int {
    var x int
    if n >= 1 {
        x = (n * fact(n-1, m))
        m[n] = x // Enable memoization
        return x
    } else {
        m[n] = 1 // Enable memoization
        return 1
    }
}

func fib(n int) int {
    if n <= 1 {
        return n
    }
    return fib(n-1) + fib(n-2)
}

func prime(x, y int) bool {
```



```

    if x < 2 {
        return false
    }
    if y == 1 {
        return true
    } else {
        if x%y == 0 {
            return false
        } else {
            return prime(x, y-1)
        }
    }
}

func main() {
    fmt.Println("Calculate primes range 0 to 100")
    for i := 0; i < 101; i++ {
        x := prime(i, i-1)
        if x == true {
            fmt.Printf("%d ", i)
        }
    }
    fmt.Println("")
    fmt.Println("Calculate Fibonacci numbers range 0 to 19")
    for i := 0; i < 20; i++ {
        x := fib(i)
        fmt.Printf("%d ", x)
    }
    fmt.Println("")
    m := make(map[int]int)
    fmt.Println("Map before looping on calling factorial function:", m)
    fmt.Println("The first time looping on factorial function it must calculate factorials")
    for i := 0; i < 10; i++ {
        x := fact(i, m)
        fmt.Printf("%d! is %d, ", i, x)
    }
    fmt.Println("")
    fmt.Println("Factorial function cached calculated factorials:", m)
    var f bool
    f = false
    fmt.Println("Second time looping some values already calculated, some are not")
    for i := 6; i < 13; i++ { // Second time looping see what is memoized and what needs calculation
        for key, value := range m { // Demonstrate memoization
            if key == i { // Previously calculated the factorial
                fmt.Printf("Memoized factorial %d! is %d\n", i, value)
                f = true
            }
        }
    }
    if f == false { // Call factorial function if value not memoized in cache

```

```
        x := fact(i, m)
        fmt.Printf("Calculated factorial %d! is %d\n", i, x)
    }
    f = false // Reset flag
}
fmt.Println("Map now:", m)
}
```

/*
Prints the following:
Calculate primes range 0 to 100
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
Calculate Fibonacci numbers range 0 to 19
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
Map before looping on calling factorial function: map[]
The first time looping on factorial function it must calculate factorials
0! is 1, 1! is 1, 2! is 2, 3! is 6, 4! is 24, 5! is 120, 6! is 720, 7! is 5040, 8! is 40320, 9! is 362880,
Factorial function cached calculated factorials: map[7:5040 8:40320 0:1 1:1 2:2 3:6 4:24 5:120 6:720
9:362880]
Second time looping some values already calculated, some are not
Memoized factorial 6! is 720
Memoized factorial 7! is 5040
Memoized factorial 8! is 40320
Memoized factorial 9! is 362880
Calculated factorial 10! is 3628800
Calculated factorial 11! is 39916800
Calculated factorial 12! is 479001600
Map now: map[9:362880 10:3628800 5:120 6:720 2:2 3:6 4:24 7:5040 8:40320 11:39916800 0:1 1:1
12:479001600]
*/

Reflection

Reflection is the ability of a program to examine and modify its own behavior and structure at runtime. It comes from one of the meanings of the word reflection where it refers to consideration or examination; it is derived from Latin via Middle English as “*reflexiōn*”.

Go uses the principle of reflection to permit the examination of variables and types, and to call their methods, without knowing the type at compile time. Reflection contains type introspection, which is the ability to examine the properties and type of an entity at runtime.

Go has a package named *reflect*. Importing this package permits implementing reflection for a program. The way the reflection is performed in the language is via the use of interfaces.

There are two important points to know when using interfaces for reflection. The first is that every entity satisfies the empty interface e.g. “*interface {}*”. Because that interface has no methods, by definition every entity satisfies it even when that entity has no methods.

The second point is that every variable that is of type interface has three values: the address of the variable in memory, the value assigned to the variable, and the type of the assigned value. Another way to think of this is that it has two value entities: the address of the variable, and a value pair containing the assigned value and the assigned value type.

The *reflect* package provides two functions that enable determining a variable’s value and its type, those functions are *reflect.ValueOf()* and *reflect.TypeOf()*. Of interest here is that the variable passed into these functions is received as a parameter specified as “*i interface{}*”. So no matter what the type passed in, be that a variable, function, or interface, it is received by these two functions as a parameter with type empty interface.

Reflection is rarely needed and should generally be avoided. However, a possible use case for reflection is when you provide a package and provide an interface in the package that needs to handle unknown data types being passed to the interface. Go uses it for some packages like *fmt* and *io* that need to handle variables of multiple data types and values, including user defined types. This is how Go handles generics, with ad hoc rather than parameterized polymorphism.

The following example code illustrates a basic use of some reflect package functions:

Reflection: Code Example

```
//Illustrating the use of reflection
package main

import (
    "fmt"
    "reflect"
)
```

```
// Demonstrate the use of reflection with reflect.TypeOf and reflect.ValueOf
// TypeOf returns the type of the value in the interface{}
// TypeOf signature is: func TypeOf(i interface{}) Type
// ValueOf returns a new value equal to the value in the interface{}
// ValueOf signature is: func ValueOf(i interface{}) Value
func main() {
    var i float32 = 3.14
    j := struct {
        firstname string
        lastname  string
    }{firstname: "Happy", lastname: "Rabbit"}
    type I interface{} // The empty interface
    var k I
    var m [3]int = [3]int{1, 2, 3}

    fmt.Println("type:", reflect.TypeOf(i)) // Get float 32variable type
    fmt.Println("value:", reflect.ValueOf(i)) // Get float32 variable value
    fmt.Println("type:", reflect.TypeOf(j)) // Get struct variable type
    fmt.Println("value:", reflect.ValueOf(j)) // Get struct variable type

    // Go into struct and get field level information from struct
    for z := 0; z < reflect.TypeOf(&j).Elem().NumField(); z++ { // Loop twice since 2 struct fields
        fieldval := reflect.ValueOf(&j).Elem().Field(z) // Get field value
        fieldtype := reflect.ValueOf(&j).Elem().Type().Field(z) // Get field name
        fmt.Printf("Struct field name: %s, struct field Value: %s\n", fieldtype.Name, fieldval)
    }
    fmt.Println("type:", reflect.TypeOf(k)) // Get interface type - will be empty
    fmt.Println("value:", reflect.ValueOf(k)) // Get interface value - will be invalid
    k = j // Assign struct to interface
    fmt.Println("type:", reflect.TypeOf(k)) // Now get interface type again, looks like struct
    fmt.Println("value:", reflect.ValueOf(k)) // Get interface value again, looks like struct
    fmt.Println("type:", reflect.TypeOf(m)) // Get array variable type
    fmt.Println("value", reflect.ValueOf(m)) // Get array variable value
}
/*
Prints the following:
type: float32
value: 3.14
type: struct { firstname string; lastname string }
value: {Happy Rabbit}
Struct field name: firstname, struct field Value: Happy
Struct field name: lastname, struct field Value: Rabbit
type: <nil>
value: <invalid reflect.Value>
type: struct { firstname string; lastname string }
value: {Happy Rabbit}
type: [3]int
value [1 2 3]
```

* /

Runes

The term rune can sometimes be confusing because it has two separate meanings. The word itself is derived from the Old Norse or Old English word “*rūn*” which means “script character”.


Rune may mean either a UTF-8 code point, or a data type, depending upon context. In the context of UTF-8, the word rune and the term UTF-8 code point are identical in meaning. Runes (UTF-8 code points) range from one byte to four bytes in length. Therefore, in this context a rune may be between one to four bytes in length, and the rune is the encoded character.

However, in the context of program code the *rune* is a data type. The size is int32, because since any UTF-8 code point may be at most four bytes, the int32 type is used for this purpose. Therefore, the word rune within a program is an **alias** for type int32, and type rune == int32.



Within program source code, the term rune always means data type int32. In conversations or in documentation, it may mean either. See also [Code Points](#).

The purpose of the rune type is to hold UTF-8 code points. It is commonly used when it is known that non-ASCII characters may need to be processed by a program, but of course it can also hold ASCII characters as well.

Rune literals are all integer values (all UTF-8 code points are integer values). But rune literals are expressed in the source code as a single character within single quotes. The compiler will convert them to the underlying UTF-8 format. Any character may be mapped to the underlying UTF-8 code point.

There is a difference between a character and a glyph. Glyphs are the physical representation of two or more characters. While any character may be expressed as a single rune, glyphs are composed of more than one character, and therefore may require more than one rune. For example, the glyph “” requires two runes to be expressed. It is also the case that a glyph might be composed of other glyphs. String literals may contain both characters and glyphs.

When printing out a rune, the verb conversion is ‘%c’. When printing out a rune literal within single quotes, for example ‘A’ or ‘表’, the verb conversion is ‘%q’. See also [UTF-8](#).

The following code shows how runes within a string literal have different lengths, and how glyphs which appear to be a single character may require two or more runes. For example, the character  is a single rune which consumes four bytes; while the ideogram  is actually a glyph that requires two runes and six bytes.

Runes: Code Example

```
// Illustrate runes and glyphs by shoring counts and lengths of runes for characters and glyphs
package main
```

```

import (
    "fmt"
    "unicode/utf8"
)

func main() {
    // Count the number of runes in a string
    // Specification: func RuneCountInString(s string) (n int)
    fmt.Println("What are the number of runes in \"a\":", utf8.RuneCountInString("a"))
    fmt.Println("What are the number of runes in \"界\":", utf8.RuneCountInString("界"))
    fmt.Println("What are the number of runes in \"𐄂\":", utf8.RuneCountInString("𐄂"))
    fmt.Println("What are the number of runes in \"得\":", utf8.RuneCountInString("得"))
    fmt.Println("What are the number of runes in \"𐄂\":", utf8.RuneCountInString("𐄂"))

    // Determine the length of a string in bytes
    fmt.Println("What is the length of the string \"a\" in bytes:", len("a"))
    fmt.Println("What is the length of the string \"界\" in bytes:", len("界"))
    fmt.Println("What is the length of the string \"𐄂\" in bytes:", len("𐄂"))
    fmt.Println("What is the length of the string \"得\" in bytes:", len("得"))
    fmt.Println("What is the length of the string \"𐄂\" in bytes:", len("𐄂"))
}
/*
This will print out the following:
What are the number of runes in "a": 1
What are the number of runes in "界": 1
What are the number of runes in "𐄂": 1
What are the number of runes in "得": 1
What are the number of runes in "𐄂": 2
What is the length of the string "a" in bytes: 1
What is the length of the string "界" in bytes: 3
What is the length of the string "𐄂" in bytes: 3
What is the length of the string "得" in bytes: 4
What is the length of the string "𐄂" in bytes: 6
*/

```

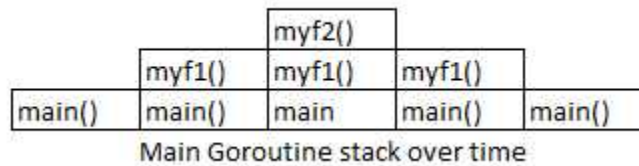
Stack

A stack is an orderly pile, and the word derives from Middle English “*stak*”. In the context of computer science, it is a simple data structure that contains “elements” of various types; it has two operations: a push which places elements onto the stack, and pop which removes elements from the stack.

In Go each goroutine has its own stack. In a simple program that uses only sequential processing, there is only one goroutine - the main goroutine. Additional goroutines are each given their own small stack to start with - a few kilobytes for each goroutine. See [Goroutines](#).

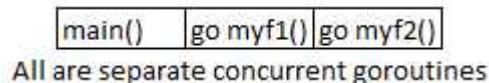
As any goroutine runs, a function call results in the called function code and its local (non-heap) variables being pushed onto the stack for that goroutine. If that first called function calls another function, then that newly called function code and its local variables are pushed onto the stack “on top of” the calling function. When a function completes, it (and its local variables) are popped from the stack, and flow of control returns to the calling function on the stack.

A view of the stack over time may be seen in the following image, which illustrates *main()* calling *myf1()*, which calls *myf2()*; *myf2()* returns flow of control to *myf1()*, which returns to *main()*.



Any variable that might be referenced outside of its lexical scope cannot be placed onto the stack, because it may have been popped off the stack by the time the program code accesses the referenced location. If this were to happen, the program would crash. Therefore, variables of this type are placed on the heap. See [Heap](#).

The following illustration shows 3 separate stacks, as the *main()* goroutine, the *myf1()* goroutine, and the *myf2()* goroutine all run concurrently. The difference is that in the above example, *myf1()* and *myf2()* are functions, whereas in the below example, *go myf1()* and *go myf2()* are goroutines. See [Concurrency](#).



Templates

A template is a pattern or gauge used to shape something, the etymology of the term is not absolutely sure but appears to come via the French “*templet*” which was derived from Latin “*templum*” which means a plank or rafter to hold something; concatenated with “*plate*” (from Middle English meaning something flat) for printing meaning to make a stereotype.

In the context of programming languages template has come to mean a document with a customized structure that can contain entered data. For Go there are two packages which support templates, *text/template* and *html/template*. Templates are executed by the program code by rendering the customized structure “as is”, while applying actions to data structures or control structures identified within the delimiters “{{” and “}}”.

Textual templates tend to be used in standard document generation systems, such as for contracts or form letters. These assemble boilerplate text that should be unchanged, while variable text may be selected by a user from a list or may be entered free form by the user. The resulting document is then assembled from the combination of the boilerplate text and the user selected or generated text.

HTML templates are generally used for creating pre-populated web pages with a mix of standard and customized information. For example, if a user is logged into a site, a page that displays the logged in user’s name would be a combination of boilerplate text but containing a variable that holds the user’s name to customize the page.

Both the text template and HTML template use the *text/template* interface, but the HTML template adds functionality to protect against code injection. Whenever the output recipient is expecting HTML, the *html/template* package should be imported and used.

For data structures any field names must be exportable, which means the first letter must be a capitalized UTF-8 character. In general practice the data structures are struct types where the type is capitalized, and the fields within the struct are also capitalized. If a struct type contains both capitalized (exported) fields as well as non-capitalized fields, and if the non-capitalized field is referenced within the “{{” “}}” action an error will be generated when the template is executed.

The following code example demonstrates use of a text template. It shows the correct usage of templates, and deliberately shows the error that results when using a non-capitalized field.

Templates: Code Example - Text Template

```
// Illustrate basic use of text template
package main
```

```
import (
    "fmt"
    "os"
```

```

    "text/template"
)

type Rabbit struct {
    Firstname string
    Lastname  string
    nonExported string // Not capitalized so not exported
    Speed    string
}

// Slow rabbit template processing
func sr() {
    p := Rabbit{Firstname: "Fat", Lastname: "Rabbit", nonExported: "Slow runner"}
    t := template.New("Template Illustration")
    t, _ = t.Parse("Welcome {{.Firstname}} {{.Lastname}}!\nYou are a {{.nonExported}}.\n")
    err := t.Execute(os.Stdout, p)
    if err != nil {
        fmt.Println("There was an error:", err)
    }
}

// Fast rabbit template processing
func fr() {
    p := Rabbit{Firstname: "Sleek", Lastname: "Rabbit", Speed: "Fast runner"}
    t := template.New("Template Illustration")
    t, _ = t.Parse("Welcome {{.Firstname}} {{.Lastname}}!\nYou are a {{.Speed}}.\n")
    err := t.Execute(os.Stdout, p)
    if err != nil {
        fmt.Println("There was an error:", err)
    }
}

func main() {
    sr()
    fr()
}

/*
Prints something like:
Welcome Fat Rabbit!
You are a There was an error: template: Template Illustration:2:12: executing "Template Illustration" at
<.nonExported>: nonExported is an unexported field of struct type main.Rabbit
Welcome Sleek Rabbit!
You are a Fast runner.
*/

```

Types

A type is a category of terms that have a common set of characteristics, these characteristics define identity for the type and the behavior of the type. The word derives from Latin “*typus*” meaning a figure or image, or from Greek “*typos*” meaning an impression or mark. In the context of computer science, the concept of types is based on type theory; and the implementation of types in each programming language is called a type system.

Go is a statically typed language with four types: basic, reference, aggregate, and interface. The types are not limited to data types, because both functions and interfaces are also types. Type conversion is permitted if it is explicit, no implicit type conversions are permitted. Perform *type assertion* and use the *type switch* to identify an unknown data type. See [Type Assertion](#).

Basic - these are numbers (integers, floating points, and complex), Booleans, and strings:

- Integers are represented by the types: int, int8, int16, int32, and int64; and unsigned integers: uint, uint8, uint16, uint32, and uint64. Type *byte* is alias for uint8; *rune* is alias for int32 and is used to denote support for UTF-8 in the code. See [Runes](#) and [UTF-8](#).
- Booleans have only two values: true and false.
- Strings are an immutable sequence of bytes. See [Immutability](#).
- Floating point numbers are of two types: float32 and float64.
- Complex numbers are of two types: complex64 (composed of a pair of float32) and complex128 (composed of a pair of float64). Complex numbers are composed of a real number (left side of decimal point) and an imaginary number (right side of decimal point).

Reference - these have their own address and a value, the value contains the address of another data value, reference types are pointers, slices, maps, channels, and functions:

- Pointer - a reference to a variable, consisting of its own address, and the address of the referenced variable; the type is pointer to variable type T, specified as “* T”.
- Slice - a reference to a variable length sequence of memory containing data elements all being the same identical type T, specified as “[] T”.
- Map - a reference to a hash table containing an unordered collection of key/value pairs; the map type is “map[K]V” where K and V are the types of keys and values; type K is restricted to types that can use the operator “==”. See [Mapping](#).
- Channel - a reference to a conduit of values of identical type T, specified as “chan T”; channels are used for communication between goroutines. See [Channels](#).
- Function - a function type is specified by its *signature* - its name, the ordered list of parameter types, and the list of return types. Two functions with the same signature have the same type. Methods are special kind of function where a “receiver type” is written between the *func* keyword and the name of the method. A given method may only be called by a variable of the specified receiver type. Unlike regular functions, multiple methods may have the same name if they differ in receiver type. See [Functions](#).

Aggregate - a concatenation of element values in memory

- Array - a fixed length sequence of n (n may be zero) elements of the same type; “[n]T”.

- Struct - implements the concept of a record which contains groups of n (n may be zero) ordered named values (called “fields”) of any type T; “*struct{}*” is an empty struct (0 fields), otherwise: “*struct{<name> T, <name> T, ..., <name> T}*”.

Interface - permits generalizing the behavior of other entities by specifying a method set e.g. “*<name of interface> interface{m1(); m2(), ..., mn()}*”; the method set may be empty, if so the interface is considered to be the empty interface and is specified as “*interface{}*”. A type “satisfies” an interface if it implements the methods of the interface. Any type satisfies the empty interface. See [Abstraction](#), [Method Set](#) and [Interface](#).

The following code example shows the usage of some Go types. See the references above to other sections for many more code examples showing the various types.

Types: Code Example

```
//Illustration of the usage of various types
package main

import (
    "fmt"
)

type Distance float64           // All variables of this type used for distance calculations only
type Temperature float32       // All variables of this type used for temperature calculations only
type Person struct {
    firstname string
    lastname  string
    age       int
} // Aggregate named type struct

func (p *Person) load(fn, ln string, a int) {
    p.firstname = fn
    p.lastname = ln
    p.age = a
} // This function type is its signature: the ordered list of parameter types and return list types

func (p *Person) show() {
    fmt.Printf("First Name: %s, Last Name: %s, Age: %d\n", p.firstname, p.lastname, p.age)
}

type shower interface {
    show() // Interface type here only has one method
} // By convention single method interface "-er"

func main() {
    var d Distance           // Declare d to be of type Distance
    var f, c Temperature    // Declare f and c to be of type Temperature
    x := 3.123 + 0.6i        // Declared and assigned type complex128 value
```

```
d = 1.5
//f = d // Runtime error: cannot use d (type Distance) as type Temperature in assignment
f = Temperature(d) // Explicitly cast type Distance to type Temperature
c = f // Both c and f are the same type Temperature
fmt.Println(d, f, c, x) // Prints: 1.5 1.5 1.5 (3.123+0.6i)
var p Person // Declare p to be of datatype Person
p.load("Jack", "Rabbit", 6)
p.show() // Call method, prints: First Name: Jack, Last Name: Rabbit, Age: 6
var i shower // Declare i to be of interface type shower
i = &p // Assign Person variable address to Shower interface variable
i.show() // Call interface, prints: First Name: Jack, Last Name: Rabbit, Age: 6
}

/*
Prints the following:
1.5 1.5 1.5 (3.123+0.6i)
First Name: Jack, Last Name: Rabbit, Age: 6
First Name: Jack, Last Name: Rabbit, Age: 6
*/
```

Type Assertion

The Go language supports a concept known as type assertion, which operates on an instance of an interface type and any other type. The notation is “*i.(T)*” where *i* is an instance of an interface type, and *T* is any other type. The purpose of type assertion is to determine whether type *T* supports type *i*.

1. If type *T* is an interface type, then determine if type *i* implements the interface *T*.
2. If type *T* is not an interface type, then the type assertion determines whether type *i* is identical to type *T*.

When the type assertion is done in an assignment statement, there are two possible outcomes. First, if type *T* matches the value in type *i*, then the value is returned. Second, if type *T* does not match the value in *i*, then there is a runtime panic. This form is: “*x := i.(T)*”

To avoid this situation, a two-argument assignment form may be used. In this case a panic does not occur. Instead either the value is assigned to the first argument and the Boolean value “true” is assigned to the second argument; or the zero type for type *T* is assigned to the first argument and the Boolean value “false” is assigned to the second argument. So: “*x, ok := i.(T)*”

Another method of performing the type assertion is to use a type switch. In a type switch, the keyword switch is used in front of the type assertion like so: “*switch i.(T)*”. Here the type assertion returns the type, not the value of the type. The case statements should be defined to match the actual type returned from the type assertion. Depending on the type returned from the assertion, then operations within the given case may be performed.

The usefulness of this capability is to enable using the empty interface when passing a value to a function where the value may be any of several types. For the receiving function to know how to properly operate on the passed in value, it needs to determine the datatype of the value. While usually the datatype of a function input parameter is specified in the function declaration, if it is desirable to permit a function to handle a given input parameter that may be of any datatype, use a combination of the empty interface with a type switch and a type assertion.

Type Assertion: Code Example

```
// Type Assertion functionality illustrated
package main

import (
    "fmt"
)

// Function demonstrates: type switch, empty interface, type assertion
func typeDetermine(v interface{}) string {
    switch v.(type) {
    case int:
        return "int" // If integer type, return "int"
```

```

    case string:
        return "string" // If string type, return string
    default:
        return "other type" // If any other type, return "other type"
    }
}

func main() {
    var value interface{} // Declare value to be empty interface
    value = "foo bar"      // Assign string to empty interface type
    // x := value.(int); fmt.Printf("%q\n", x) // Don't do this, type assertion fails!
    // Runtime error prints: panic: interface conversion: interface {} is string, not int
    str, ok := value.(string) // Type assertion form: x, ok := v.(T)
    if ok {
        fmt.Printf("value is: %q\n", str) // Prints value is: "foo bar"
    } else {
        fmt.Printf("value is not a string\n")
    }
    str = typeDetermine(value) // Call to determine value type
    fmt.Printf("value: %v, type: %q\n", value, str) //value: foo bar, type: "string"
    value = 1000 // Assign integer to empty interface type
    str = typeDetermine(value) // Call to determine value type
    fmt.Printf("value: %v, type: %q\n", value, str) // value: 1000, type: "int"
    value = 12.345 // Assign float to empty interface type
    str = typeDetermine(value) // Call to determine value type
    fmt.Printf("value: %v, type: %q\n", value, str) // value: 12.345, type: "other type"
    var a [3]int
    value = a
    switch val := value.(type) { // Another example of type switch usage, falls to default
    case int:
        fmt.Printf("value is an int, holds: %v.\n", val)
    case float64:
        fmt.Printf("value is a float64, holds: %v.\n", val)
    default:
        fmt.Printf("value is some other type, holds: %v.\n", val)
    }
}
/*
Prints the following:
value is: "foo bar"
value: foo bar, type: "string"
value: 1000, type: "int"
value: 12.345, type: "other type"
value is some other type, holds: [0 0 0].
*/

```

UTF-8

UTF-8 is an acronym for Unicode Transformation Format - 8 bit. It is a form of character encoding that can specify 1,112,064 separate characters (valid code points) within Unicode using from one to four 8-bit bytes. Every UTF-8 code point (rune) is unique. There are no duplicate numerical sequences within the UTF-8 code space. See [Code Points](#).

It is backwards compatible with ASCII, and the first 128 bytes of the UTF-8 address space are the ASCII character set. Each ASCII character is encoded within a single 8-bit octet.

In most cases when dealing with the English language and basic mathematics, all characters within the source code will be ASCII characters. However, the Go language treats the source code as UTF-8. This means all known characters can be represented in the Go source code - all human languages, all mathematical characters, all diacritical marks, and even many pictographic symbols (emojis and ideograms). Most of the Unicode space is empty and is being populated with new symbols as they are discovered or created.

Glyphs require two or more characters (runes) to express. Many glyphs are not yet represented in the UTF-8 code space. For example, there is an area requested in the Unicode space for Mayan glyphs (from U+15500 to U+159FF), however no detailed proposal has yet been submitted, which means these glyphs are not yet in UTF-8. They will almost certainly be allocated as graphic runes. The following image set shows three public domain images of the many known Mayan glyphs.



Egyptian hieroglyphs have been assigned the range of U+13000 to U+1342F which can contain 1,072 code points. 1,039 Egyptian hieroglyphs have been assigned. For example, the following hieroglyph exists at U+13000 (integer value 77824).



To be able to copy a hieroglyph into code, the full range of Egyptian hieroglyphs may be found here: <https://unicode-table.com/en/#egyptian-hieroglyphs>. The above Egyptian hieroglyph code space location, and other information, can be shown by the following example code, although the hieroglyph itself within the code looks very small, and will look like `𐀀`.

UTF-8: Code Example

```
// Show some UTF-8 codespace information about an Egyptian hieroglyph
```



```
package main

import (
    "fmt"
    "unicode"
    "unicode/utf8"
)

func main() {
    const h = '𐀀'
    fmt.Printf("%+q\n", h)
    if unicode.IsGraphic(h) {
        fmt.Println("This is a graphic rune")
    }
    fmt.Println("Length of hieroglyph is:", utf8.RuneLen(h))
    fmt.Println("The rune is displayed as:", string(h))
}
/*
Prints the following:
'\U00013000'
This is a graphic rune
Length of hieroglyph is: 4
The rune is displayed as: 𐀀
*/
```

Appendix I: Some Comments

This section contains some advice for good programming practices with the language and points out what might be unexpected language features.

As a general good programming practice, put nothing in your main package except for initializations, creation of variables needed for goroutines, calls to launch and manage goroutines, control actions on those goroutines, and closure and stoppage calls on shared variables.

All struct data types, methods with signatures dependent on those struct data types, and interfaces satisfied by those methods, should go into either a single package or in functionally specific packages containing those struct data types, struct methods, and interfaces. Access to structs based on the struct data types should be through interfaces satisfied by the structs. There are several reasons for this, but a key reason is to prevent circular dependencies between packages. Otherwise package A may need interfaces from package B, and conversely, so they want to import each other. Instead put the structs, struct methods, and interfaces in package C, then have both A and B import C, and no more circular dependency.

Restrict all accesses to structs except via interfaces. An easy way to do this is to not capitalize structs or the struct methods, only the interfaces. This ensures encapsulation of the data and enforces control over how the data may be manipulated.

Usually if a function (or method, or goroutine) can generate an error, it should return an error. Because functions can return multiple parameters, an error parameter can be returned in addition to the result. When importing the *error* package, return nil as the error value when there is no error.

The underscore character '' is quite useful in several situations. When using the iota generator, it is used to skip the numerical sequence, for example, to ensure the sequence starts with 1 instead of 0. It may serve as an operand on the left-hand side of an assignment in cases where the value is to be ignored. It may precede a package name in an import declaration if the goal is to benefit from package initializations only, while not importing any of the package contents. It can also be used to test whether a type satisfies an interface without actually using the interface (in those cases where type checking happens at runtime and not compile time).

I hope you have enjoyed this book. I hope you have found the concept of laying out computer science conceptual areas, and then describing them in the context of the Go language.

Best regards to all,
John Tullis

Appendix II: Additional Code Examples

Demonstrate use of error values from user developed functions

//Illustrate use of error values

package main

import (

 "errors"

 "fmt"

)

func mySqrt(f float64) (float64, error) {

 if f < 0 {

 return 0, errors.New("math: square root of negative number")

 } else {

 return (f * f), nil // Return nil if no problem

 }

}

func main() {

 var f float64

 f = 2.5

 if ret, err := mySqrt(f); err == nil { // Multi-valued return

 fmt.Printf("Square root of %v is %v\n", f, ret)

 } else {

 fmt.Printf("Received error on %v: %s\n", f, err)

 }

 f = -2.5

 if ret, err := mySqrt(f); err == nil { // Multi-valued return

 fmt.Printf("Square root of %v is %v\n", f, ret)

 } else {

 fmt.Printf("Received error on %v: %s\n", f, err)

 }

}

/*

Printed results:

Square root of 2.5 is 6.25

Received error on -2.5: math: square root of negative number

*/

Simple way to load a map with keys and values

//Illustrate map loading, detecting an existing key, and ranging over a map

package main

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

```
func main() {
    mymap := map[string]struct{}{} // Declare a map
    i := "Dog"
    j := "Cat"
    k := "Buffalo"

    // If no key in a map, assign something
    if _, ok := mymap[i]; !ok { // Ignore value, just want to know if there is a key
        mymap[i] = struct{}{} // No key, so assign key and value
    }
    if _, ok := mymap[j]; !ok { // Ignore value, just want to know if there is a key
        mymap[j] = struct{}{} // No key, so assign key and value
    }
    if _, ok := mymap[k]; !ok { // Ignore value, just want to know if there is a key
        mymap[k] = struct{}{} // No key, so assign key and value
    }
    if _, ok := mymap[k]; !ok { // Ignore value, just want to know if there is a key
        mymap[k] = struct{}{} // No key, so assign key and value
    } else {
        fmt.Println("Key already exists", mymap[k])
    }

    for fn, val := range mymap { // Iteration order over the map is indeterminate!
        fmt.Println("KV Pair: ", fn, val)
    }
}
```

```
/*
Prints the following:
Key already exists {}
KV Pair: Dog {}
KV Pair: Cat {}
KV Pair: Buffalo {}
*/
```

Demonstrates one way to convert a struct to a map

//Illustrate converting a struct to a map via use of JSON marshalling/unmarshalling
package main

```
import (
    "encoding/json"
    "fmt"
)

type MyD struct { // All field names must be capitalized for JSON!
    Firstname string
    Lastname  string
    Weight    int
    Age       int
}

// Convert a struct into a map
func main() {
    rabbit := &MyD{ // Declare a rabbit
        Firstname: "Fast",
        Lastname:  "Rabbit",
        Age:       25,
    } // Weight value will be null value for type int

    var myI map[string]interface{} // Declare a map
    bytes, _ := json.Marshal(rabbit) // Pass in rabbit as interface{}, return contents as []byte
    json.Unmarshal(bytes, &myI)    // Pass in bytes as []byte, unmarshal into myI as interface{}

    // iterate through map
    for fieldname, val := range myI { // Iteration order over the map is indeterminate!
        fmt.Println("KV Pair: ", fieldname, val)
    }
}

/*
Printed results like (remember order is indeterminate in maps):
KV Pair: Weight 0
KV Pair: Age 25
KV Pair: Firstname Fast
KV Pair: Lastname Rabbit
*/
```

Demonstrates why closing parentheses are needed for anonymous functions

//Illustrate anonymous function usage

package main

import (

 "fmt"

)

func main() {

 i, err := func() (int, error) {

 fmt.Println("Foo")

 return fmt.Println("Bar") // Returns number of bytes written, and error value

 }() // The parentheses are required to actually call the function

 fmt.Printf("%v %v\n", i, err)

 x := func() (int, error) {

 fmt.Println("Fast")

 return fmt.Println("Rabbit") // Returns number of bytes written, and error value

 } // No parentheses needed here, not actually calling the function

 i, err = x() // See the parentheses here also, needed to call the function

 fmt.Printf("%v %v\n", i, err)

}

/*

Prints the following:

Foo

Bar

4 <nil>

Foo

Bar

*/

Demonstrate a struct with a function field

```
//Illustrate usage of a struct with a function field
package main
```

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

```
type s struct {
    f func(name string) string
}
```

```
func main() {
    x := s{f: func(name string) string {
        return "Foo " + name
    }}
    fmt.Println(x.f("Bar"))
}
```

```
/*
Prints the following:
Foo Bar
*/
```

Demonstrating the use of maps where the values are structs

```
//Illustrate map containing structs
package main

import (
    "fmt"
)

type s struct {
    a string
    b int
}

func main() {
    m := make(map[string]*s)
    m["Foo"] = &s{}
    m["Foo"].a = "abc"
    m["Foo"].b = 1
    m["Bar"] = &s{}
    m["Bar"].a = "xyz"
    m["Bar"].b = 2
    fmt.Println("map m entry at key Foo is:", m["Foo"])
    fmt.Println("map m entry at key Bar is:", m["Bar"])
}

/*
Prints the following:
map m entry at key Foo is: &{abc 1}
map m entry at key Bar is: &{xyz 2}
*/
```


Demonstrate generation of pseudorandom map keys and values

//Illustrate usage of pseudorandom map keys and values

```
package main
```

```
import (
    "fmt"
    "math/rand"
)
```

```
func main() {
    var r = []rune("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ")
    a := make([]rune, 5)
    m := make(map[string]int, 5)
    for i := 0; i < 5; i++ {
        s := func() string {
            for i := range a {
                a[i] = r[rand.Intn(len(r))]
            }
            return string(a)
        }()
        m[s] = rand.Intn(10000) // Load map m with pseudorandom string keys and values
    }
    fmt.Println(m)
}
```

```
/*
```

Prints something like the following:

```
map[XVlBz:1318 baiCM:8511 AjWwh:1445 Hctcu:6258 xhxKQ:3015]
```

```
*/
```

Demonstrate tree creation, loading, traversal

```
//Illustrate usage of a tree with recursive data insertion and tree traversal
package main
```

```
import (
    "fmt"
    "math/rand"
)
```

```
// This will be the root of the tree
type Tree struct {
    root *node
}
```

```
// Tree will contain nodes
type node struct {
    data int
    left *node
    right *node
}
```

```
func (t *Tree) Insert(data int) {
    if t.root == nil {
        t.root = &node{data: data}
    } else {
        t.root.insert(data)
    }
}
```

```
// Move through tree recursively inserting data
func (n *node) insert(data int) {
    if data <= n.data {
        if n.left == nil {
            n.left = &node{data: data}
        } else {
            n.left.insert(data)
        }
    } else {
        if n.right == nil {
            n.right = &node{data: data}
        } else {
            n.right.insert(data)
        }
    }
}
```

```
// Walk through the tree recursively showing data
func walk(n *node) {
    if n == nil {
        return
    }
}
```

```

    }
    walk(n.left)
    fmt.Println("Node has value:", n.data)
    walk(n.right)
}

func main() {
    var t Tree
    var n *node
    fmt.Println("Load the tree")
    for i := 0; i < 10; i++ {
        v := rand.Intn(75) * rand.Intn(75)
        fmt.Println("Loading:", v) // Show loaded value
        t.Insert(v)
    }
    //Prepare to walk the tree
    n = t.root
    fmt.Println("Traverse the tree")
    walk(n)
}
/*
Prints something like:
Load the tree
Loading: 672
Loading: 2773
Loading: 558
Loading: 1625
Loading: 0
Loading: 2684
Loading: 168
Loading: 2072
Loading: 720
Loading: 3472
Traverse the tree
Node has value: 0
Node has value: 168
Node has value: 558
Node has value: 672
Node has value: 720
Node has value: 1625
Node has value: 2072
Node has value: 2684
Node has value: 2773
Node has value: 3472
*/

```

Sorting a list of planet names not using built in sort function

```
//Illustrating custom sort function
```

```
package main
```

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

```
var listOfPlanets []string = []string{ //Unsorted
    "Jupiter",
    "Mars",
    "Mercury",
    "Pluto",
    "Neptune",
    "Earth",
    "Venus",
    "Saturn",
    "Uranus",
}
```

```
func printPlanetsList(slice []string) {
    for i := 0; i < len(slice); i++ {
        fmt.Println(slice[i])
    }
}
```

```
func main() {
    fmt.Println("Unsorted planets list")
    printPlanetsList(listOfPlanets)
    fmt.Println()
    listlength := len(listOfPlanets)

    for i := 0; i < listlength; i++ { //Sort the planets
        for j := 0; j < listlength-1; j++ {
            if listOfPlanets[j] > listOfPlanets[j+1] {
                temp := listOfPlanets[j]
                listOfPlanets[j] = listOfPlanets[j+1]
                listOfPlanets[j+1] = temp
            }
        }
    }
    fmt.Println("Planets sorted alphabetically by name")
    printPlanetsList(listOfPlanets)
}
```

```
/*
```

```
Prints the following:
```

```
Unsorted planets list
```

```
Jupiter
```

```
Mars
```

Mercury
Pluto
Neptune
Earth
Venus
Saturn
Uranus

Planets sorted alphabetically by name

Earth
Jupiter
Mars
Mercury
Neptune
Pluto
Saturn
Uranus
Venus
*/

Client Server Two Way Communication

// Illustrates two way communication between a server and multiple clients

package main

import (

 "fmt"

 "log"

 "net"

 "time"

)

// Client goroutine

func simulateClient1() {

 // Connect to a server

 conn, err := net.Dial("tcp", "127.0.0.1:8000")

 if err != nil {

 log.Fatal(err)

 }

 // Ensure server disconnect when done

 defer conn.Close()

 // Communicate with the server

 serverResponse := make([]byte, 128)

 for i := 1; i < 6; i++ { // Loop 5 times

 conn.Write([]byte(fmt.Sprintf("Client 1 sent message #%d", i)))

 _, err := conn.Read(serverResponse)

 if err != nil {

 log.Fatal(err)

 }

 fmt.Println("Server response: ", string(serverResponse))

 time.Sleep(1000 * time.Millisecond)

 }

 fmt.Println("Client 1 exiting")

}

func simulateClient2() {

 // Connect to a server

 conn, err := net.Dial("tcp", "127.0.0.1:8000")

 if err != nil {

 log.Fatal(err)

 }

 // Ensure server disconnect when done

 defer conn.Close()

 // Communicate with the server

 serverResponse := make([]byte, 128)

 for i := 1; i < 4; i++ { // Loop 3 times

 conn.Write([]byte(fmt.Sprintf("Client 1 sent message #%d", i)))

 _, err := conn.Read(serverResponse)

```
        if err != nil {
            log.Fatal(err)
        }
        fmt.Println("Server response: ", string(serverResponse))
        time.Sleep(1000 * time.Millisecond)
    }
    fmt.Println("Client 1 exiting")
}

// Handle the client connection for the server
func connHandler(conn net.Conn) {
    // Defer, but guarantee, to close the client connection
    defer conn.Close()

    i := 0
    result := make([]byte, 128)
    for {
        n, _ := conn.Read(result)
        if n == 0 { // No more incoming messages
            break
        } else {
            fmt.Println("Client request: ", string(result))
            i++
            conn.Write([]byte(fmt.Sprintf("Sending server receipt #%d", i)))
        }
    }
    fmt.Println("Server connection handler exiting")
}

// Server goroutine
func simulateServer() {
    // Create a server connection
    svrconn, err := net.Listen("tcp", "127.0.0.1:8000")
    if err != nil {
        log.Fatal(err)
    }
    // Defer, but guarantee, to close the server connection
    defer svrconn.Close()

    for {
        // Accept connection from client request
        conn, err := svrconn.Accept()
        if err != nil {
            log.Fatal(err)
        }
        // Let connection handler deal with client
        go connHandler(conn)
    }
}
```

```
func main() {
    go simulateServer() // Launch a server simulator
    go simulateClient1() // Launch a server client simulator
    go simulateClient2() // Launch a server client simulator

    // Give goroutines time to run
    time.Sleep(10 * (1000 * time.Millisecond))
    fmt.Println("Main exiting")
}
/*
Prints something like:
Client request: Client 1 sent message #1
Server response: Sending server receipt #1
Client request: Client 2 sent message #1
Server response: Sending server receipt #1
Client request: Client 2 sent message #2
Server response: Sending server receipt #2
Client request: Client 1 sent message #2
Server response: Sending server receipt #2
Client request: Client 1 sent message #3
Server response: Sending server receipt #3
Client request: Client 2 sent message #3
Server response: Sending server receipt #3
Client request: Client 2 exiting
Server connection handler exiting
Client request: Client 1 sent message #4
Server response: Sending server receipt #4
Client request: Client 1 sent message #5
Server response: Sending server receipt #5
Client request: Client 1 exiting
Server connection handler exiting
Main exiting
*/
```