

Introduction to Programming in Go

A photograph of a large-scale data center or server room. The perspective is looking down a central aisle between two long rows of server racks. The racks are densely packed with hardware, and numerous small blue and yellow lights are visible, indicating active components. The ceiling above is dark with visible structural beams and lighting fixtures.

1. Getting started with Go

Preliminaries

Introductions

1. Your area of expertise.
2. Your current job role and responsibilities.
3. Programming experience.
4. Expectations for the course.

Expectations

1. Go is installed on your computer if you are not using a supplied virtual machine.
2. You can program in a C-type language.

Hello World

Hello World

As required by international law... first program is Hello World

```
// Example 01-01 Hello World

package main

import "fmt"

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

Compiling and Running

- "go run" compiles and executes a Go program.
- Executable is deleted right after execution.

```
[Module01]$ ls  
Ex01-01.go  
[Module01]$ go run Ex01-01.go  
Hello World!  
[Module01]$ ls  
Ex01-01.go
```

Building

- "go build" compiles packages and dependencies.
- executable exists after command finishes.

```
[Module01]$ ls  
Ex01-01.go  
[Module01]$ go build Ex01-01.go  
[Module01]$ ls  
Ex01-01 Ex01-01.go  
[Module01]$ ./Ex01-01  
Hello World!  
[Module01]$
```

The Go Tool

Usage:

```
go command [arguments]
```

The commands are:

build	compile packages and dependencies
clean	remove object files
doc	show documentation for package or symbol
env	print Go environment information
fix	run go tool fix on packages
fmt	run gofmt on package sources
generate	generate Go files by processing source
get	download and install packages and dependencies
install	compile and install packages and dependencies
list	list packages
run	compile and run Go program
test	test packages
tool	run specified go tool
version	print Go version
vet	run go tool vet on packages

Go File Structure - Main Package

```
// Example 01-01 Hello World
```

```
package main
```

```
import "fmt"

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

- packages in Go are libraries or modules.
- each source file begins with a package declaration.
- the main package is not a library but declares the file to be a standalone executable program.

Go File Structure - Imports

```
// Example 01-01 Hello World

package main

import "fmt"

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

- to use symbols in other packages, the packages must be imported.
- it is a compile time error to import a package that is not used.
- package names serve as name spaces.

Go File Structure - *main()*

```
// Example 01-01 Hello World

package main

import "fmt"

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

- the `main()` function is always in the `main` package.
- execution of an application begins with a call to `main()`.

Go File Structure - Imported Symbol

```
// Example 01-01 Hello World

package main

import "fmt"

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

- imported symbols are prefixed with the package name.
- only symbols that start with a capital letter can be imported.
- symbols that start with a lowercase letter are private to a package.

Go File Structure - Multiple Imports

```
// Example 01-02 Hello World

package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println(strings.ToUpper("Hello World!"))
}
```

Why Go?

Large Scale Software Development

The Go programming language was conceived in late 2007 as an answer to some of the problems we were seeing developing software infrastructure at Google.

The problems introduced by multicore processors, networked systems, massive computation clusters, and the web programming model were being worked around rather than addressed head-on.

Moreover, the scale has changed: today's server programs comprise tens of millions of lines of code, are worked on by hundreds or even thousands of programmers, and are updated literally every day. To make matters worse, build times, even on large compilation clusters, have stretched to many minutes, even hours.

Rob Pike

Rationale for Go

1. Not a "better" programming language or new way of programming.
2. C and C++ produce faster executing code in general
3. Addresses issues of build and delivery for complex and large scale software projects.
4. Utilizes modern technology for efficiency which older languages can not do.
5. Address issues of scale that cannot be addressed with older languages' compile and build technology.
6. Provides a consistent tool set to avoid third party tool mahem.

Complexity Factors

Factors that contribute to complexity in builds

1. Slow compilation and dependency resolution.
2. Lines of code and number of modules.
3. Number of programmers.
4. Multiple third party tools inconsistently used.
5. Different subsets of language used.
6. Older languages unable to use modern hardware efficiently.

Reductionistic Go

Go simplifies for build efficiency and clean code

1. Features that make other languages overly complex are omitted.
2. Features producing overly complex compiled code are eliminated.
3. Features that slow compilation are eliminated.
4. Concurrency is a fundamental part of the language.
5. Incorporates remote importing of packages.
6. Supports modern environments and practices (eg. unit testing).

Formatting Go

Go Formatting

Go is formatted much like C or Java except that:

1. Semi-colons are required only to separate statements on the same line – eols ('\n') are used to end a statement.
2. We can not insert eols freely like we can in Java or C.
3. Opening braces cannot be preceded by an eol.
4. Other white space is not significant.
5. "gofmt file" applies proper Go formatting to the file.
6. "gofmt" does not fix errors due to eol issues.

Standard Libraries

Standard Libraries

1. Like other languages, a large number of standard libraries like "fmt" exist.
2. We will be mentioning them and using them from time to time in the code.
3. There is extensive documentation on line at the Go language site.
4. Aside from pointing out specific features of interest, you are expected to explore the libraries on your own initiative.

Lab 1: Getting Started

Introduction to Programming in Go

A photograph of a large-scale data center or server room. The perspective is looking down a central aisle between two long rows of server racks. The racks are densely packed with hardware, and numerous blue and green LED lights are visible, glowing from the front panels and internal components of the servers. The ceiling above is dark and features a complex network of metal beams, pipes, and overhead lighting fixtures. The floor is a light-colored polished concrete.

2. Variables and Basic Data Types

Module Topics

1. Built-in or basic data types
2. Variable declarations
3. Variable scope
4. Constants
5. Pointers

Basic Data Types

Basic Data Types

Data Types

Numeric				Non-numeric	
integer		floats	complex	string	bool
signed	unsigned	float32 float64	complex64 complex128		
int8	uint8				
int16	uint16				
int32	uint32				
int64	uint64				
int - either int32 or int64				rune alias for int32 byte alias for uint8	
uint - either uint32 or uint64					

About Basic Data Types

1. **int** is either **int32** or **int64** depending on architecture.
2. Similarly, **uint** is either **uint32** or **uint64**.
3. Strings are immutable, like in Java.
4. Only explicit conversions are allowed.
5. No mixed mode calculations are allowed.

Variables

Default Initialization to Zero Value

```
// Example 02-01 Declaring Variables - Zero Inits
package main

import "fmt"

var x float32
var c complex64
var b bool
var str string

func main() {
    var message string = "x=%f c=%f b=%t str=%s i=%d \n"
    var i int
    fmt.Printf(message, x, c, b, str, i)
}
```

```
[Module02]$ go run ex02-01.go
x=0.000000 c=(0.000000+0.000000i) b=false str="" i=0
```

Explicit Initialization

```
// Example 02-02 Initializing Variables
package main

import "fmt"

var x float32 = 32.0
var c complex64 = 5.0 + 3.1i
var b bool = true
var str string = "Hi"

func main() {
    var message string = "x=%f c=%f b=%t str=%s i=%d \n"
    var i int = 3
    fmt.Printf(message, x, c, b, str, i)
}
```

```
[Module02]$ go run ex02-02.go
x=32.000000 c=(5.000000+3.100000i) b=true str=|Hi| i=3
```

Implicit Initialization

```
// Example 02-03 Implicit Initialization
package main

import "fmt"

var x = 32.0
var c = 5.0 + 3.1i
var b = true
var str = "Hi"

func main() {
    var message = "x=%T c=%T b=%T str=%T i=%T \n"
    var i = 3
    fmt.Printf(message, x, c, b, str, i)
}
```

```
[Module02]$ go run ex02-03.go
x=float64 c=complex128 b=bool str=string i=int
```

Implicit Initialization

```
// Example 02-04 Implicit Initialization
package main

import "fmt"

var x = float32(0)
var c = complex64(0)
var b = true
var str = "Hi"

func main() {
    var message = "x=%T c=%T b=%T str=%T i=%T \n"
    var i = uint8(0)
    fmt.Printf(message, x, c, b, str, i)
}
```

```
[Module02]$ go run ex02-04.go
x=float32 c=complex64 b=bool str=string i=uint8
```

Variations on a Theme

```
// Example 02-05 Variations
package main

import "fmt"

var i, j int8
var b, str, x = true, "Hi", float32(45)

func main() {
    fmt.Printf("i=%T j=%T b=%T str=%T x=%T \n",
        i, j, b, str, x)
}
```

```
[Module02]$ go run ex02-05.go
i=int8 j=int8 b=bool str=string x=float32
```

Declaration Short Form

```
// Example 02-06 Short Form

package main

import "fmt"

var (
    k int = 1
    m int
)

func main() {
    i, j, k := 3, 4, 5
    x := 1.2
    fmt.Println("i=%d j=%d x=%f k=%d \n", i, j, x, k)
}
```

```
[Module02]$ go run ex02-06.go
i=3 j=4 x=1.200000 k=5
```

Variable Scope

Variable Scope - Package Scope

```
// Example 02-07 Variable Scope
```

```
package main
```

```
import "fmt"
```

```
var k int = 1
```

k has Package Scope

```
func main() {
```

```
    i := 1
```

```
{
```

```
    x := 1.2
```

```
    fmt.Println("i=%d d x=%f k=%d \n", i, x, k)
```

```
}
```

```
}
```

```
[Module02]$ go run ex02-07.go
i=1 x=1.200000 k=1
```

Variable Scope - Local Function Scope

```
// Example 02-07 Variable Scope
```

```
package main
```

```
import "fmt"
```

```
var k int = 1
```

```
func main() {
```

```
    i := 1
```

```
{
```

```
    x := 1.2
```

```
    fmt.Println("i=%d d x=%f k=%d \n", i, x, k)
```

```
}
```

```
}
```

i has Function Scope

```
[Module02]$ go run ex02-07.go
i=1 x=1.200000 k=1
```

Variable Scope - Local Block Scope

```
// Example 02-07 Variable Scope

package main

import "fmt"

var k int = 1

func main() {
    i := 1
    {
        x := 1.2
        fmt.Println("i=%d d x=%f k=%d \n", i, x, k)
    }
}
```

x has Block Scope

```
[Module02]$ go run ex02-07.go
i=1 x=1.200000 k=1
```

Variable Scope - Shadowing

```
// Example 02-08 Shadowing

...
var k int = 1

func main() {
    fmt.Println("Package Scope k=%d \n", k)
    k := 2
    fmt.Println("Function Scope k=%d \n", k)
    {
        k := 3
        fmt.Println("Block Scope k=%d \n", k)
    }
    fmt.Println("Function Scope k=%d \n", k)
}
```

```
[Module02]$ go run ex02-08.go
Package Scope k=1
Function Scope k=2
Block Scope k=3
Function Scope k=2
```

Constants

Constants

```
// Example 02-09 Constants  
...  
const a float32 = 1  
const b = 4 / 3  
  
func main() {  
    const c string = "Hello Word"  
    fmt.Printf("a=%T b=%T c=%T\n", a, b, c)  
    fmt.Printf("a=%f b=%d c=%s\n", a, b, c)  
}
```

```
[Module02]$ go run ex02-09.go  
a=float32 b=int c=string  
a=1.000000 b=1 c=Hello Word
```

Enums with iota

1. `iota` is a built in enumerator for building sets of enumerations.
2. `iota` starts at 0 and increments by 1 for each constant.
3. An expression to generate iota can be used instead of 0.
4. `iota` is reset to 0 each time we use it with a new set of constants.
5. Any constant initialized with `iota` is of type int.
6. The blank `_` symbol can be used to make `iota` start at 1.

Enums with iota

```
// Example 02-10 Enums with iota
...
const (
    a = iota
    b
    c
)

func main() {
    fmt.Printf("a=%T b=%T c=%T\n", a, b, c)
    fmt.Printf("a=%f b=%d c=%s\n", a, b, c)
}
```

```
[Module02]$ go run ex02-10.go
a=int b=int c=int
a=0 b=1 c=2
```

Enums with iota - Blank Variable

```
// Example 02-11 Enums with iota and _

...
const (
    _ = iota
    a
    b
    c
)

func main() {
    fmt.Printf("a=%T b=%T c=%T\n", a, b, c)
    fmt.Printf("a=%f b=%d c=%s\n", a, b, c)
}
```

```
[Module02]$ go run ex02-11.go
a=int b=int c=int
a=1 b=2 c=3
```

Enums with iota - Generator Expression

```
// Example 02-12 Enums with Generator  
...  
const (  
    a = iota * 2  
    b  
    c  
)  
  
func main() {  
    fmt.Printf("a=%T b=%T c=%T\n", a, b, c)  
    fmt.Printf("a=%d b=%d c=%d\n", a, b, c)  
}
```

```
[Module02]$ go run ex02-12.go  
a=int b=int c=int  
a=0 b=2 c=4
```

Pointers

Pointers

1. Pointers work just like in C or C++.
2. Pointer to an int would be declared as "var p *int".
3. Address-of operator is "&".
4. `p := &i` would assign a pointer p the address of i.
5. De-referencing operator is "*".
6. `*p` gives the contents of the location pointed to by p.
7. Pointer assignments and comparisons "==" are allowed.
8. Pointer arithmetic is not allowed.

Pointers

```
// Example 02-13 Pointers  
...  
  
func main() {  
    var i int = 187  
    var p *int  
  
    p = &i  
    fmt.Println("i=", i, " &i or p =", p, " *p =", *p)  
    *p = -12  
    fmt.Println("i=", i, " &i or p =", p, " *p =",  
    *p)  
}
```

```
[Module02]$ go run ex02-13.go  
i= 187 &i or p = 0xc82000a398 *p = 187  
i= -12 &i or p = 0xc82000a398 *p = -12
```

Pointers

Variable	Address	Value	
i	df65ac	187	i := 187
p	d367ff	df65ac	p := &i
i	df65ac	-12	*p := -12
p	d367ff	df65ac	

Lab 2: Variables

Introduction to Programming in Go

A photograph of a large-scale data center or server room. The perspective is looking down a central aisle between two long rows of server racks. The racks are densely packed with hardware, and numerous blue and green LED lights are visible, glowing from the front panels and internal components of the servers. The ceiling above is dark and features a complex network of metal beams and structural supports. The floor is a light-colored polished concrete.

3. Go Program Control Structures

Module Topics

1. Operators
2. **if** statements
3. **for** loops
4. **switch** statements

Operators

Go Operator Differences

1. Standard C type operator behavior except for....
2. No mixed mode operations.
3. No exponent operator.
4. Increment/decrement (`x++` and `x--`) are statements, not expressions.
5. Prefix notation not allowed - ie. `++x` is illegal.
6. Multiple assignments are done in parallel.

Mixed Mode Operations

```
// Example 03-01 Operators
package main

import "fmt"

func main() {
    a, b, c := uint8(1), uint16(1), int8(1)
    fmt.Println("1. a + b =", a + b)
    fmt.Println("2. a + c =", a + c)
    fmt.Println("3. a + uint8(b)=", a + uint8(b))
    fmt.Println("4. uint8(a) + c=", int8(a) + c)
}
```

```
[Module03]$ go run ex03-01.go
./ex03-01.go:10: invalid operation: a + b (mismatched types uint8 and uint16)
./ex03-01.go:11: invalid operation: a + c (mismatched types uint8 and int8)
```

Mixed Mode Operations

```
// Example 03-01 Operators
package main

import "fmt"

func main() {
    a, b, c := uint8(1), uint16(1), int8(1)
    //fmt.Println("1. a + b =", a + b)
    //fmt.Println("2. a + c =", a + c)
    fmt.Println("3. a + uint8(b)=", a + uint8(b))
    fmt.Println("4. uint8(a) + c=", int8(a) + c)
}
```

```
[Module03]$ go run ex03-01.go
3. a + uint8(b)= 2
4. uint8(a) + c= 2
```

Parallel Assignment

```
// Example 03-02 Parallel Assignment
package main

import "fmt"

func main() {
    first, last := "York", "New"
    fmt.Println(first, last)
    first, last = last, first
    fmt.Println(first, last)
}
```

```
[Module03]$ go run ex03-02.go
York New
New York
```

Conditionals - if statements

Conditionals - Differences in Go

1. Standard C type conditional behavior except for....
2. No parentheses "(..)" around test condition.
3. Local variables can be defined in the `if` statement itself.
4. Braces "{ .. }" are mandatory for all `then` and `else` blocks.
5. Opening "{" for each block cannot start on a new line.
6. The `else` keyword cannot appear at the start of a new line.

Basic Conditional Statement

```
// Example 03-03 Basic if Statement
package main

import "fmt"

func main() {
    x := 22
    if x == 0 {
        fmt.Printf("%d is zero\n", x)
    } else if x % 2 == 0 {
        fmt.Printf("%d is even\n", x)
    } else {
        fmt.Printf("%d is odd\n", x)
    }
}
```

```
[Module03]$ go run ex03-03.go
22 is even
```

Local Variables

```
// Example 03-04 Local Variables in Conditionals
package main

import "fmt"

func main() {
    if x , y := 22, "hi" ; x == 0 {
        fmt.Println("Value of x=", x, " y=", y)
    } else if x % 2 == 0 {
        fmt.Println("Value of =", x, " y=", y)
    } else {
        fmt.Println("Value of x=", x, " y=", y)
    }
}
```

```
[Module03]$ go run ex03-04.go
Value of x= 22  y= hi
```

Variable Assignment

```
// Example 03-05 Non-local variables in Conditionals
package main

import "fmt"

var x = 1
var y = "bye"

func main() {
    if x, y = 22, "hi" ; x % 2 {
        fmt.Println("Value of x=", x, " y=", y)
    } else {
        fmt.Println("Value of x=", x, " y=", y)
    }
}
```

```
[Module03]$ go run ex03-05.go
Value of x= 22  y= hi
```

For Loops

Loops - Differences in Go

1. The only loop construct in Go is the **for** loop.
2. Can function as a while loop.
3. No parentheses "(..)" allowed in the for clause.
4. Braces "{..}" are mandatory for the loop body.
5. The pre and post terms in the for clause can be empty.

Basic For Loop

```
// Example 03-06 Basic for loop

package main

import "fmt"

func main() {
    var total = 0
    for count := 0; count < 100; count++ {
        total += count
    }
    fmt.Println("total = ", total)
```

```
[Module03]$ go run ex03-06.go
total = 4950
```

Basic For Loop - Multiple Variables

```
// Example 03-07 Basic for loop
....  
  
func main() {
    var total = 0
    for i, m := 0, "abort" ; i < 100; i++ {
        total += i
        if total > 100 {
            fmt.Println(m)
            break
        }
    }
    fmt.Println("total = ", total)
}
```

```
[Module03]$ go run ex03-07.go
abort
total = 105
```

For Loop - Non-local Variables

```
// Example 03-08 Non-local Variables
....  
  
func main() {
    var total, i = 1000, 1000  
  
    for i, total := 0, 0 ; i < 100; i++ {
        if total > 200 {
            continue
        } else {
            total += i
        }
        fmt.Println("total = ", total)
    }
}
```

```
[Module03]$ go run ex03-08.go
total = 205
```

For Loop as a While Loop

```
// Example 03-09 For Loop as While Loop
.....
func main() {
    count := 0

    for count < 2 {
        count++
    }
    fmt.Println("count = ", count)
}
```

```
[Module03]$ go run ex03-09.go
count = 2
```

Looping Using Ranges

```
// Example 03-10 Looping Using Range
.....
func main() {
    test := "Hi!"

    for i, c := range test {
        fmt.Printf("Letter %d is %#U\n", i, c)
    }
}
```

```
[Module03]$ go run ex03-10.go
Letter 0 is U+0048 'H'
Letter 1 is U+0069 'i'
Letter 2 is U+0021 '!'
```

Switch Statement

Switch - Differences in Go

1. Go **switch** statements are more general.
2. Test values can be any expression, not just integers.
3. Cases break automatically unless the **fallthrough** keyword is used.
4. The **break** statement breaks out of a clause at any point.
5. Test values are optional, cases will execute on a true result.

Simple Switch Statement

```
// Example 03-11 Simple Switch Statement
...
func main() {
    for i := 0; i < 3; i++ {
        switch i {
        case 0:
            fmt.Println("Case 0")
        case 1:
            fmt.Println("Case 1")
            fallthrough
        default:
            fmt.Println("Default")
    }
}
```

```
[Module03]$ go run ex03-11.go
Case 0
Case 1
Default
Default
```

Switch Statement Break

```
// Example 03-12 Switch Statement Break
...
func main() {
    for i := 0; i < 3; i++ {
        switch i {
        case 0:
            fmt.Println("Case 0")
        case 1:
            fmt.Println("Case 1")
            break
            fmt.Println("After break")
        default:
            fmt.Println("Default")
    }
}
```

```
[Module03]$ go run ex03-12.go
Case 0
Case 1
Default
```

Switch - Non-integral Test Value

```
// Example 03-13 Non-integral Test
...
func main() {
    os := "fedora"

    switch os {
    case "fedora", "redhat":
        fmt.Println("Open Source")
    case "Windows":
        fmt.Println("Proprietary")
    default:
        fmt.Println("unknown")
    }
}
```

```
[Module03]$ go run ex03-13.go
Open Source
```

Switch - No Test Value

```
// Example 03-14 No Test Value
...
func main() {
    x := 22

    switch {
    case x == 0:
        fmt.Println("zero")
    case x % 2 == 0:
        fmt.Println("even")
    default:
        fmt.Println("odd")
    }
}
```

```
[Module03]$ go run ex03-14.go
even
```

Lab 3: Control Structures

Introduction to Programming in Go

A photograph of a large-scale data center or server room. The perspective is looking down a central aisle between two long rows of server racks. The racks are densely packed with hardware, and numerous blue and yellow lights are visible, glowing from the front panels and internal components of the servers. The ceiling above is dark and features a complex network of metal beams and structural supports. The floor is a light-colored polished concrete.

4. Basic Functions in Go

Module Topics

1. Function syntax
2. Multiple return values
3. Deferred execution
4. Recursion
5. Varadic functions
6. Error handling
7. Panics and recoveries

Function Syntax

Basic Function Syntax

1. Function form with unnamed return value:

```
func fname(var1 type1, ...varn typen) type { ... }
```

2. Function form with named return value:

```
func fname(var1 type1, ...varn typen) (var type) { ... }
```

3. Named return values are initialized to their zero values.
4. Named return values are used like normal variables in the function.
5. Unnamed values require an explicit **return** argument list.
6. Named return values only need a naked return statement.

Basic Function Form

```
// Example 04-01 Basic function syntax
...
func square(x int) int {
    y := x * x
    return y
}

func main() {
    x := 4
    retval := square(x)
    fmt.Println("x = ", x, "square = ", retval)
}
```

```
[Module04]$ go run ex04-01.go
x = 4 square = 16
```

Named Return Value

```
// Example 04-02 Named Return Value
...
func square(x int) (result int) {
    y := x * x
    return
}

func main() {
    x := 4
    fmt.Println("x = ", x, "square = ", square(x))
}
```

```
[Module04]$ go run ex04-02.go
x = 4 square = 16
```

Multiple Parameters

```
// Example 04-03 Multiple Parameters
...
func divides(x, y int) (div bool) {
    div = (y % x) == 0
    return y
}

func main() {
    x, y := 2, 21
    fmt.Println(x, "|", y, " is ", divides(x, y))
}
```

```
[Module04]$ go run ex04-03.go
2 | 21  is  false
```

Multiple Return Values

Multiple Return Values

1. Function form with multiple return values:

```
func fname(v1 t1...) (type1, type2, ...) { ... }
```

2. Function form with multiple named return values:

```
func fname(v1 t1...) (var1 type1, var2 type2) { ... }
```

3. Same rules apply as for single return values.

Multiple Return Values

```
// Example 04-04 Multiple return values
...
func divides(x, y int) (int, int) {
    return (y/x), (y%x)
}

func main() {
    x, y := 3, 23
    quote, rem := divides(x,y)
    fmt.Println(x, "/", y, " is ", quote, "R", rem)
}
```

```
[Module04]$ go run ex04-04.go
3 / 23  is  7 R 2
```

Multiple Named Return Values

```
// Example 04-05 Multiple named return values
...
func divides(x, y int) (int q, int r) {
    q = y / x
    r = y % x
    return
}

func main() {
    x, y := 3, 23
    quote, rem := divides(x,y)
    fmt.Println(x, "/", y, " is ", quot, "R", rem)
}
```

```
[Module04]$ go run ex04-05.go
3 / 23  is  7 R 2
```

Deferred Execution

Deferred Execution

1. Function execution is deferred using the `defer` operator:

```
defer fname(x,y)
```

2. Defers function execution until the calling function is about to exit.
3. Parameters are evaluated when called, not when executed.
4. Multiple functions can be deferred: executed in a LIFO order.
5. Often used for cleanup and recovery from panics.

Simple Defer

```
// Example 04-06 Deferred execution  
...  
func f(message string) {  
    fmt.Println(message)  
}  
  
func main() {  
    m := "before defer"  
    defer f(m)  
    m = "after defer"  
    fmt.Println(m)  
}
```

```
[Module04]$ go run ex04-06.go  
after defer  
before defer
```

Stacked Defer

```
// Example 04-07 Stacked defer
...
func f(k int) {
    fmt.Println("executing f(%d)\n", k)
}

func main() {
    for i := 1; i < 4; i++ {
        fmt.Printf("called f(%d)\n", i)
        defer f(i)
    }
    fmt.Println("end main")
}
```

```
[Module04]$ go run ex04-07.go
called f(1)
called f(2)
called f(3)
end main
executed f(3)
executed f(2)
executed f(1)
```

Recursion

Recursion

```
// Example 04-08 Recursion
...
func sum(k int) int {
    fmt.Println("executing sum(%d)\n", k)
    if k > 0 {
        return k + sum(k-1)
    } else {
        return 0
    }
}

func main() {
    fmt.Println(sum(3))
}
```

```
[Module04]$ go run ex04-08.go
executed sum(3)
executed sum(2)
executed sum(1)
executed sum(0)
6
```

Varadic Functions

Varadic Functions

1. Varadic functions take a variable number of parameters

```
func fname(msg string, x ... int)
```

2. Ellipsis indicates variable number of parameters.
3. Only the last parameter can use ellipsis.
4. Varadic parameters are loaded into an array of the parameter type.

Varadic Functions

```
// Example 04-09 Varadic functions
...
func addup(nums ... int) (sum int) {
    for _, val := range nums {
        sum += val
    }
    return
}

func main() {
    fmt.Println(sum(3))
}
```

```
[Module04]$ go run ex04-09.go
15
```

Error Handling

Error Handling

1. Go does not use exceptions to handle errors.
2. When a function fails, an error object is created and returned.
3. The error object must be the last return value in the return value list.
4. When no error occurs, the error object is nil.
5. When an error occurs, any return values are considered invalid.
6. Calling function then tests for the error occurrence.
7. Errors are not created automatically, the program has to create them.
8. A simpler form is the "comma ok" idiom with bools instead of errors.

Error Handling

```
// Example 04-10 Error Handling
...
import "errors"

func division(num, denom int) (int, error) {
    if denom == 0 {
        return 0, errors.New("Divide by zero")
    }
    return (num/denom), nil
}

func main() {
    res, e := division(56, 0)
    if e != nil {
        fmt.Println(e)
    } else {
        fmt.Println(res)
    }
}
```

```
[Module04]$ go run ex04-10.go
Divide by zero
```

Error Handling

```
// Example 04-11 Error Handling
...
import "errors"

func division(num, denom int) (int, error) {
    if denom == 0 {
        return 0, errors.New("Divide by zero")
    }
    return (num/denom), nil
}

func main() {
    res, e := division(56, 2)
    if e != nil {
        fmt.Println(e)
    } else {
        fmt.Println(res)
    }
}
```

```
[Module04]$ go run ex04-11.go
```

28

Comma OK Idiom

```
// Example 04-12 Comma OK Idiom

...
func division(num, denom int) (int, bool) {
    if denom == 0 {
        return 0, false
    }
    return (num/denom), true
}

func main() {
    res, ok := division(56, 2)
    if ok {
        fmt.Println(res)
    }
}
```

```
[Module04]$ go run ex04-12.go
28
```

Panics and Recoveries

Panics and Recoveries

1. A panic is generated by Go when a runtime error occurs.
2. During a panic, all deferred functions are called, then execution halts.
3. Panics can be generated by calling the `panic()` function.
4. Panics can be recovered from by executing the `recover()` function.
5. The `recover()` function only works in deferred functions.
6. Panics should be used only for critical problems, use errors otherwise.
7. The `recover()` function returns the panic item, nil otherwise.

Runtime Panic

```
// Example 04-13 Runtime Panic  
...  
func main() {  
    x, y := 1, 0  
    fmt.Println(x / y)  
}
```

```
[Module04]$ go run ex04-13.go  
panic: runtime error: integer divide by zero  
[signal 0x8 code=0x1 addr=0x40102c pc=0x40102c]  
(stacktrace omitted)  
exit status 2
```

Generated Panic

```
// Example 04-14 Generated Panic  
...  
func division(num, denom int) int {  
    if denom == 0 {  
        panic("Dividing by zero?!?")  
    }  
    return (num/denom)  
}  
  
func main() {  
    res := division(56, 0)  
    fmt.Println(res)  
}
```

```
[Module04]$ go run ex04-14.go  
panic: Dividing by zero?!?  
(stacktrace omitted)  
exit status 2
```

Panic and Recovery

```
// Example 04-15 Panic and Recovery
...
func rec() {
    r := recover()
    if r != nil {
        fmt.Println("recovered value = ", r)
    }
}

func main() {
    x, y := 1, 0
    defer rec()
    fmt.Println(x / y)
}
```

```
[Module04]$ go run ex04-15.go
recovered value =  runtime error: integer divide by zero
```

Lab 4: Functions

Introduction to Programming in Go

A photograph of a large-scale data center or server room. The space is filled with floor-to-ceiling server racks, all illuminated from within by a multitude of small blue lights, creating a glowing, futuristic atmosphere. The racks are arranged in long, narrow rows that recede into the distance towards a central point. The ceiling above is dark and features a complex network of overhead pipes, beams, and structural supports. A single bright light fixture hangs down from the center of the ceiling. The floor is a polished, light-colored tile.

5. Arrays and Slices

Module Topics

1. Arrays
2. Slices

Arrays

Arrays in Go

1. Go arrays are similar to other C-Style programming languages.
2. Fixed length numbered sequence of elements of a single type.
3. The length is part of the array type eg. `[3]int` and `[5]int` are types.
4. Array types are always one-dimensional.
5. Arrays may be combined to form multi-dimensional types.
6. The length of the array can be found using `len(array)`.
7. Array elements are accessed as in Java or C with `[...]` operator.
8. Arrays are value objects like `int` and `float32`, not reference objects.

Basic Array Syntax

1. Arrays are defined using the syntax:

`var name [size] type`

`var a [3]int` defines an array of type `[3]int`

2. By default all elements are initialized to their zero values.
3. Access to array elements is the same as in other C-style languages.
4. Attempt to access elements out of range generates a panic.

Basic Array Syntax

```
// Example 05-01 Basic array syntax  
.  
  
func main() {  
    var a [3]int  
    fmt.Println("a =", a)  
    a[0] = 1  
    a[1] = a[0] + 1  
    fmt.Println("a =", a)  
}
```

```
[Module05]$ go run ex05-01.go  
a = [0 0 0]  
a = [1 2 0]
```

Explicit Array Initialization

1. Arrays can be initialized with array literals:

```
var ar = [5]int{3: 3, 4: 4}
```

2. Using [...] for makes the compiler count the literal values for the size.
3. Some of the elements can be initialized using the syntax:

```
var ar = [size]type{index1 : value1, ... indexn: valuen}
```

4. All other elements are initialized to their zero value.
5. If [...] is used with the index:value syntax as in point 3, the size is the last index referenced.

Explicit Array Initialization

```
// Example 05-02 Explicit array initialization  
...  
  
func main() {  
    var ar1 = [5]int{0, 1, 2, 3, 4}  
    var ar2 = [...]string{"Hello", "World"}  
  
    ar3 := [2]bool{true, false}  
    ar4 := [...]int{3: -1, 4: -1}  
  
    fmt.Println("ar1=", ar1, "length=", len(ar1))  
    fmt.Println("ar2=", ar2, "length=", len(ar2))  
    fmt.Println("ar3=", ar3, "length=", len(ar3))  
    fmt.Println("ar4=", ar4, "length=", len(ar4))  
}
```

```
[Module05]$ go run ex05-02.go  
ar1= [0 1 2 3 4] length= 5  
ar2= [Hello World] length= 2  
ar3= [true false] length= 2  
ar4= [0 0 0 -1 -1] length= 5
```

Array Operations

1. Arrays of the same type can be compared using `==` and `!=`
2. Array assignment copies the array from the RHS to LHS.
3. Arrays are passed by value as function call arguments.
4. Iteration over arrays is done with the `range` operation.

Array Comparison

```
// Example 05-03 Array comparison  
...  
func main() {  
    var ar1 = [5]int{0, 1, 2, 3, 4}  
    var ar2 = [5]int{0, 1, 2, 3, 4}  
  
    fmt.Println("ar1 == ar2 is", ar1 == ar2))  
    fmt.Println("ar1 != ar2 is", ar1 != ar2))  
}
```

```
[Module05]$ go run ex05-03.go  
ar1 == ar2 is true  
ar1 != ar2 is false
```

Arrays as Function Parameters

```
// Example 05-04 Array as parameter  
...  
func delta(prm [3]int) {  
    prm[0] = -1  
    fmt.Println("prm = ", prm)  
}  
  
func main() {  
    var arg = [3]int{99, 98, 97}  
    fmt.Println("arg = ", arg)  
    delta(arg)  
    fmt.Println("arg = ", arg)  
}
```

```
[Module05]$ go run ex05-04.go  
arg =  [99 98 97]  
prm =  [-1 98 97]  
arg =  [99 98 97]
```

Array Assignment

```
// Example 05-05 Array assignment  
...  
func main() {  
    var ar1 = [3]int{99, 98, 97}  
    var ar2 [3]int  
    ar2[0] = 0  
    fmt.Println("ar1 =", ar1)  
    fmt.Println("ar2 =", ar2))  
}
```

```
[Module05]$ go run ex05-05.go  
ar1 = [99 98 97]  
ar2 = [0 98 97]
```

Array Iteration with range

```
// Example 05-06 Iteration with range  
...  
func main() {  
    words := [...]string{99, 98, 97}  
    for index, value := range words {  
        fmt.Println(index, " ", value)  
    }  
}
```

```
[Module05]$ go run ex05-06.go  
0 the  
1 best  
2 of  
3 times
```

Multidimensional Arrays

1. Multi-dimensional arrays are built up in layers.
2. For example a two dimensional array would be defined as either

[2][2]int = { {1,2},{3,4}}

or

[2][2]int = {[2]int {1,2},[2]int{3,4}}

3. Ragged arrays (ie. non-rectangular) are not allowed.
4. All of the usual rules for arrays still apply.

Multidimensional Arrays

```
// Example 05-07 Multidimensional array  
.  
  
func main() {  
    var matrix [2][3]  
    value := 10  
    for row, col := range matrix {  
        for index, _ := range col {  
            matrix[row][index] = value  
            value++  
        }  
    }  
    fmt.Println("Matrix: ",matrix)  
}
```

```
[Module05]$ go run ex05-07.go  
Matrix: [[10 11 12] [13 14 15]]
```

Multidimensional Initialization

```
// Example 05-08 Multidimensional initialization  
...  
  
func main() {  
    var matrix [4][4]int{  
        [4]int{1, 2, 3, 4}  
        [4]int{2, 4, 8, 16}  
        [4]int{3, 9, 27, 81}  
        [4]int{4, 16, 64, 256}  
    }  
    fmt.Println("Matrix: ", matrix)  
}
```

```
[Module05]$ go run ex05-08.go  
Matrix: [[1 2 3 4] [2 4 8 16] [3 9 27 81] [4 16 64 256]]
```

Multidimensional Initialization

```
// Example 05-09 Multidimensional initialization  
...  
func main() {  
    var matrix [4][4]int{  
        {1, 2, 3, 4}  
        {2, 4, 8, 16}  
        {3, 9, 27, 81}  
        {4, 16, 64, 256}  
    }  
    fmt.Println("Matrix: ", matrix)  
}
```

```
[Module05]$ go run ex05-09.go  
Matrix: [[1 2 3 4] [2 4 8 16] [3 9 27 81] [4 16 64 256]]
```

Slices

Slices

1. A slice is a reference to a contiguous segment of an underlying array.
2. A slice has a start position, a length and a capacity.
3. Slices can be thought of as sub-sequences of arrays.
4. Slices look and act syntactically just like arrays.
5. Slices are dynamic – their length can change.
6. Slice capacity is how big the slice can become.
7. Slices are used more in Go code than arrays.

Slices

```
// Example 05-10 Slices  
...  
func main() {  
  
    var a = [6]int{0, 1, 2, 3, 4, 5}  
    s := a[2:] // s is a slice of the array a  
    fmt.Println("s= ", s)  
  
    a[4] = -20 // changing underlying array  
    fmt.Println("s= ", s)  
  
    s[0] = 999 // change the array via the slice  
    fmt.Println("a= ", a)  
}
```

```
[Module05]$ go run ex05-10.go  
s= [2 3 4 5]  
s= [2 3 -20 5]  
a= [0 1 999 3 -20 5]
```

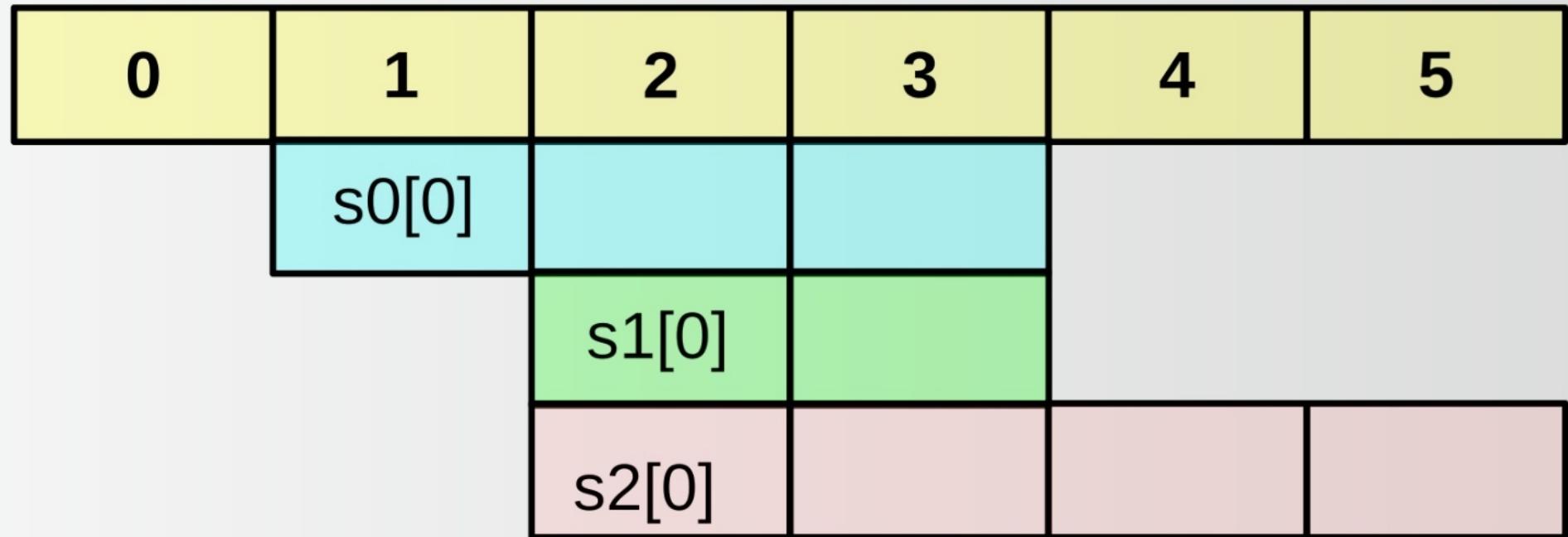
Slices of slices

```
// Example 05-11 Slices  
...  
func main() {  
    var a = [...]int{0, 1, 2, 3, 4, 5}  
    s0 := a[1:4]  
    s1 := a[1:3]  
    s2 := a[0:4]  
    fmt.Println("s0 length=", len(s0), " s1 length=",  
                len(s1), " s2 length=", len(s2))  
    fmt.Println("s0 =", s0, " s1=", s1, " s2 =", s2)  
}
```

```
[Module05]$ go run ex05-11.go  
s0 length= 3  s1 length= 2  s2 length= 4  
s0= [1 2 3]  s1= [2 3]  s2= [2 3 4 5]
```

Slices of slices

a



Creating Slices Directly

1. We can create slices directly in two ways:
2. i) Using an array literal

`[]type{list of elements}`

3. Key point [...] defines array, [] defines slice.
4. ii) Using make()
`s := make([]type,len)` where len is initial length.
5. In both cases, Go makes an anonymous underlying array.
6. The array can only be accessed through the slice.

Creating Slices Directly

```
// Example 05-12 Slices  
...  
func main() {  
    s1 := []int{1, 2, 3}  
  
    s2 := make([]int, 3)  
  
    fmt.Println(s1, s2)  
}
```

```
[Module05]$ go run ex05-12.go  
[1 2 3] [0 0 0]
```

Appending Elements

1. append() function adds elements to the end of a slice.

slice := append(slice,item)

2. If appending exceeds capacity of slice, the slice is re-sized.
3. Re-sizing doubles the capacity of the slice.

Appending Elements

```
// Example 05-13 Appending Slices  
...  
  
func main() {  
    a := [...]int{100, 200, 300}  
    s := a[:2]  
    fmt.Println("Initially a=", a, "s= ", s)  
    s = append(s, -1) //result is a[2] == -1  
    s[0] = 0           // result a[0] == 0  
    fmt.Println("After first op a=", a, "s=", s)  
    s = append(s, -2) // this would go in a[3]?  
    fmt.Println("After second op a=", a, "s=", s)  
    s[0] = 999 // a now remains unchanged  
    fmt.Println("After third op a=", a, "s=", s)  
}
```

```
[Module05]$ go run ex05-13.go  
Initially a= [100 200 300] s= [100 200]  
After first op a= [0 200 -1] s= [0 200 -1]  
After second op a= [0 200 -1] s= [0 200 -1 -2]  
After third op a= [0 200 -1] s= [999 200 -1 -2]
```

Specifying Initial Capacity

```
// Example 05-14 Initial Capacity  
...  
func main() {  
  
    s1 := make([]int, 1, 3)  
    for i := 0; i < 10; i++ {  
        s1 = append(s1, i)  
        fmt.Println("s1=", s1, "len=", len(s1),  
                    "Cap=", cap(s1))  
    }  
}
```

```
[Module05]$ go run ex05-14.go  
s1= [0 0] len= 2 Cap= 3  
s1= [0 0 1] len= 3 Cap= 3  
s1= [0 0 1 2] len= 4 Cap= 6  
s1= [0 0 1 2 3] len= 5 Cap= 6  
s1= [0 0 1 2 3 4] len= 6 Cap= 6  
s1= [0 0 1 2 3 4 5] len= 7 Cap= 12  
s1= [0 0 1 2 3 4 5 6] len= 8 Cap= 12  
s1= [0 0 1 2 3 4 5 6 7] len= 9 Cap= 12  
s1= [0 0 1 2 3 4 5 6 7 8] len= 10 Cap= 12  
s1= [0 0 1 2 3 4 5 6 7 8 9] len= 11 Cap= 12
```

Slices as Function Arguments

```
// Example 05-15 Slice as Function Argument
...
func f(p []int) {
    p[0] = -1
}

func main() {
    s := []int{1, 2, 3, 4}
    fmt.Println("Before call", s)
    f(s)
    fmt.Println("After call", s)
}
```

```
[Module05]$ go run ex05-15.go
Before call [1 2 3 4]
After call [-1 2 3 4]
```

Lab 5: Arrays and Slices

Introduction to Programming in Go

A photograph of a large-scale data center or server room. The space is filled with floor-to-ceiling server racks, all illuminated from within by a multitude of small blue lights, creating a glowing, futuristic atmosphere. The racks are arranged in long, narrow aisles that recede into the distance towards a bright overhead light fixture. The ceiling is dark and features a complex network of metal beams and structural supports. The overall scene conveys a sense of massive computing power and infrastructure.

6. Maps

Module Topics

1. Maps in Go
2. Declaring and Creating Maps
3. Working with Map Entries
4. Map Iteration

Maps in Go

Maps in Go Basics

1. A map in Go is a reference to an underlying hash table.
2. Map elements are (key, value) pairs declared as

```
var map1 map[keytype]valuetype
```

3. Maps have types defined by combining the key type and value type.
4. Key type must have == and != operators defined.
5. Valuetype can be any Go type.
6. Maps are fast – access in near constant time.
7. Maps are passed by reference to functions.

Declaring and Creating Maps

Declaring and Creating Maps

1. Maps are created using the `make()` function.
2. Maps can also be created by providing a map literal.
3. `map[key] = value` overwrites value if the key already exists.
4. `map[key] = value` adds value if the key does not exist.
5. Duplicate keys in a map literal are not allowed.
6. Maps can be declared with an initial capacity.

Creating Maps with Literals

```
// Example 06-01 Creating maps with literals
...
var errs map[int]string

func main() {
    severity := map[string]string{
        "Blue":    "normal",
        "Orange": "moderate",
        "Red":     "severe"}
    severity["Black"] = "apocalyptic"

    fmt.Println(severity, " size =", len(severity))
    fmt.Println(errs, " size =", len(errs))
}
```

```
[Module06]$ go run ex06-01.go
map[Blue:normal Orange:moderate Red:severe Black:apocalyptic]
size = 4
map[]  size = 0
```

Creating Maps with make()

```
//Example 06-02 Creating maps with make()

...
var errs map[int]string

func main() {
    errs = make(map[int]string)
    errs[0] = "Hardware"
    errs[1] = "Segmentation fault"
    fmt.Println(errs, " size =", len(errs))
    errs[0] = "Firmware fault"
    fmt.Println(errs, " size =", len(errs))
    fmt.Println("The errorcode '0' is a ", errs[0])
}
```

```
[Module06]$ go run ex06-02.go
map[1:Segmentation fault 0:Hardware]  size = 2
map[0:Firmware fault 1:Segmentation fault]  size = 2
The errorcode '0' is a  Firmware fault
```

Empty versus nil Maps

```
// Example 06-03 Empty versus nil maps
...
var errs map[string]string

func main() {
    fmt.Println("Check for nil before make()", errs == nil)
    fmt.Println("Length of errs ", len(errs))

    errs = make(map[string]string)

    fmt.Println("Check for nil after make()", errs == nil)
    fmt.Println("Length of errs ", len(errs))
}
```

```
[Module06]$ go run ex06-03.go
Check for nil before make()  true
Length of errs  0
Check for nil after make()  false
Length of errs  0
```

Map with Initial Capacity

```
// Example 06-04 Map capacity  
...  
var errs map[string]string  
func main() {  
    errs = make(map[string]string, 200)  
    fmt.Println("Length of errs ", len(errs))  
}
```

```
[[Module06]$ go run ex06-4.go  
Length of errs  0
```

Working with Map Entries

Working with Map Entries

1. `map[key]` always returns a value.
2. If the key does not exist, the default zero of the value type is returned.
3. The more complete form is
`value, bool = map[key]`
4. The bool value is often assigned to the variable “`ok`”.
5. If “`ok`” is false, the key is not in the map.
6. Attempting to delete a non-existent element causes a panic.

The Comma ok Idiom

```
// Example 06-05 comma ok update
...
func update(m map[int]int, key int, val int) {
    _, ok := m[key]
    if ok {
        m[key] = val
    }
}

func main() {
    data := map[int]int{1: 100, 3: 300}
    update(data, 1, -1)
    update(data, 2, 200)
    fmt.Println(data)
}
```

```
[Module06]$ go run ex06-05.go
map[1:-1 3:300]
```

Map Iteration

Map Iteration

```
// Example 06-06 Iteration
...
func main() {
    dotw := map[string]int{
        "Sun": 1, "Mon": 2, "Tue": 3, "Wed": 4,
        "Thu": 5, "Fri": 6, "Sat": 7}
    for day, num := range dotw {
        fmt.Printf("(%s : %d) ", day, num)
    }
}
```

```
[Module06]$ go run ex06-0c.go
(Thu : 5) (Fri : 6) (Sat : 7) (Sun : 1) (Mon : 2) (Tue : 3) (Wed : 4)
```

Lab #6: Maps

Introduction to Programming in Go

A photograph of a large-scale data center or server room. The perspective is looking down a central aisle between two long rows of server racks. The racks are densely packed with hardware, and numerous blue and yellow lights are visible, glowing from the front panels and internal components of the servers. The ceiling above is dark and features a complex network of metal beams and structural supports. The floor is a light-colored polished concrete.

7. Advanced Functions in Go

Module Topics

1. Functions as first class objects
2. Function Literals
3. Anonymous Functions
4. Closures

Functions as First Class Objects

Basic Function Syntax

1. Functions are first class objects in Go.
2. Functions can be assigned to variables.
3. Functions can be passed to and returned from other functions.
4. A function type is number and types of parameters and return values:

Eg. func f(x, y, string)(a , b int){...} has type

func(string,string)(int, int)

Function Variables

```
// Example 07-01 Function variables

func f1() string {
    return "I'm f1"
}

func f2() string {
    return "and I'm f2"
}

func main() {
    f := f1
    fmt.Printf("f is of type '%T' \n", f)
    fmt.Println(f())
    f = f2
    fmt.Println(f())
}
```

```
[example 07] go run ex07-01.go
f is of type 'func() string'
I'm f1
and I'm f2
```

Function as Parameter

```
// Example 07-02 Functions as parameters
...
func f1() string {
    return "I'm f1"
}
func f2(fparam func() string) {
    fmt.Println(fparam())
}

func main() {
    f := f1
    f2(f)
}
```

```
[Module07]$ go run ex07-02.go
I'm f1
```

Function as Return Value

```
// Example 07-03 Function as return value
...
func f1() string {
    return "I'm f1"
}
func f2() func() string {
    return f1
}

func main() {
    f := f2()
    fmt.Println(f())
}
```

```
[Module07]$ go run ex07-03.go
I'm f1
```

Function Literals

Function Literal

1. We can assign a function body to a variable.
2. In this case the function body is called a "function literal."
3. Acts like any other type of literal (first class object).
4. Can be passed as parameters, etc.
5. Function literals are identified by the func keyword.

Function Literal

```
// Example 07-04 Function literal  
.  
  
func main() {  
    f := func(i int) bool { return i == 0 }  
    fmt.Println("2 == 0 is", f(2))  
    fmt.Println("0 == 0 is", f(0))  
}
```

```
[Module07]$ go run ex07-04.go  
2 == 0 is false  
0 == 0 is true
```

Function Literal as Parameter

```
// Example 07-05 Function parameter

...

func f2(p func(int) bool) {
    fmt.Println("2 == 0 is", p(2))
    fmt.Println("0 == 0 is", p(0))
}

func main() {
    f2(func(i int) bool { return i == 0 })
}
```

```
[Module07]$ go run ex07-05.go
2 == 0 is false
0 == 0 is true
```

Anonymous Functions

Anonymous Functions

1. Anonymous functions are function literals that are executed without being assigned to a variable.

`func() { ... }` is a function literal

`func() {...}()` executes the function literal (anon function)

2. The following assigns the function literal to the variable f

`f := func() {...}`

4. The following assigns the result of executing the function to g

`g := func() {...}()`

5. The second of these is an anonymous function since there no way to reference the function itself.

Inner Function

```
// Example 07-06 Executing a literal directly  
.  
  
func main() {  
    z := func(x int) (y int) {  
        y = x + 1  
        return  
    }(0)  
    fmt.Println(z)  
}
```

```
[Module07]$ go run ex07-06.go  
1
```

Anonymous Inner Function

```
// Example 07-07 Anonymous inner function

...

func main() {
    defer func(name string) {
        fmt.Println("Hello ", name)
        return
    }("World")
    fmt.Println("Main Function")
}
```

```
[Module07]$ go run ex07-07.go
Main Function
Hello World
```

Closures

Closure

1. Functions can have “free variables” – variables used in the scope of the function that are not defined in the function.
2. When we create an instance of the function, we also create copies of any of the free variables referenced in the function body.
3. The combination of function instance plus its copies of the free variables is called a closure.

```
a := 1
f: = func () int{
    return a + 1
}
```

4. In the example above, "a" is a free variable and we will need to remember its value to execute f()

Simple Closure

```
// Example 07-08 Simple Closure

...

func f(p func(string)) {
    p("Mars") // "a" is out of scope here
}

func main() {
    a := "again "
    z := func(name string) {
        fmt.Println("Hello ", a, name)
        return
    }
    f(z)
}
```

```
[Module07]$ go run ex07-08.go
Hello again Mars
```

Another Closure Example

```
// Example 07-09 Closure
// From the go tour on the golang website
...

func intSeq() func() int {
    i := 0
    return func() int {
        i += 1
        return i
    }
}

func main() {
    nextInt := intSeq()
    fmt.Println(nextInt())
    fmt.Println(nextInt())
    newInts := intSeq()
    fmt.Println(newInts())
}
```

```
[Module07]$ go run ex07-09.go
1
2
1
```

Lab 7: Advanced Functions

Introduction to Programming in Go

8. Types and Structs

Module Topics

1. User Defined Types
2. Defining Structs
3. Working with Structs
4. Embedding Structs

User Defined Types

User Defined Types

1. We can create new types by using the `type` keyword.
2. Aliases allow us to create "clones" of existing types.
3. We can create complex data types using type.
4. Structs in Go work much like structs in C and C++.
5. Structs are part of how we implement OO in Go.

Aliases

1. Aliases allow us to create a copy of a type. For example, Go defines:

```
type rune int32  
type byte uint8
```

- 2 An alias belongs to the package that it is defined in.
3. Aliases are their own type, not just an alternate name.
4. Useful when we define methods in the next module .

Defining Structs

Defining Structs

1. Structs are defined using the following syntax:

```
type sname struct {  
    field1 type1  
    field2 type2  
    ...  
}
```

2. Structs are types so variables can be defined to be the type corresponding to a struct definition:

```
var v1 sname  
var v2 sname
```

Creating Structs

```
// Example 08-01 Basic struct definition  
...  
type point struct{ x, y int }  
type employee struct {  
    fname, lname string  
    id            int  
    job           string  
    salary        int          }  
  
func main() {  
    var anil employee  
    var p point  
    fmt.Println("anil=", anil, "p=", p)  
}
```

```
[Module058$ go run ex08-01.go  
anil= { 0 0} p= {0 0}]
```

Initializing Structs

1. We can use literals to initialize structs:

```
var anil = employee {"Anil", "Patel", 8971,  
                     "Developer", 100000}  
  
p := point{4,2}
```

2. Selected fields may be initialized:

```
var anil = employee{id: 8971, fname: "Anil",  
                     lname: "Patel"}
```

3. If field names are provided, the order does not matter.
4. Any field not named in the initial list is set to its default zero value.

Initialization of Structs

```
// Example 08-02 Struct initialization  
...  
type point struct{ x, y int }  
type employee struct { ... }  
  
func main() {  
    var p = point{2, 3}  
    fmt.Println("p =", p)  
  
    anil := employee{"Anil", "Patel", 9891,  
                    "Developer", 100000}  
    greta := employee{id: 8897, fname: "Greta",  
                     lname: "Smith"}  
  
    fmt.Println("anil =", anil)  
    fmt.Println("greta =", greta)  
}
```

```
[Module08]$ go run ex08-02.go  
p = {2 3}  
anil = {Anil Patel 9891 Developer 100000}  
grea = {Greta Smith 8897 0}
```

Using New

1. new() is used to allocate memory for a struct.
2. new() returns a pointer to the newly created object.

```
var greta *employee = new(employee)
```

3. The underlying object is accessed with the de-referencing operator *.
4. Short way for calling the new operator and then initializing the fields of the new point object:

```
p := &point{4, 5}
```

Using New

For Clarity:

`bob := employee{0,0}` bob is an object

`ptr_to_bob := &bob` ptr_to_bob is the address of bob

`ptr_to_sue := &employee{fname: "Sue"}`

`*(ptr_to_sue)` is the sue object pointed to by ptr_to_sue

Using New

```
// Example 08-03 Using new  
.  
type point struct{ x, y int }  
  
func main() {  
  
    var pp1 *point = new(point) // pp1 is a pointer  
    fmt.Println("pp1 =", pp1, "*pp=", *pp1)  
  
    p := point{3, 4} // p is a variable  
    pp := &p // pp is a pointer  
    fmt.Println("pp =", pp, "*pp=", *pp, "p=", p)  
  
    pp2 := &point{5, 6} //pp2 is a pointer  
    fmt.Println("pp2 =", pp2, "*pp2=", *pp2)  
}
```

```
[Module08]$ go run ex08-03.go  
pp1 = &{0 0} *pp= {0 0}  
pp = &{3 4} *pp= {3 4} p= {3 4}  
pp2 = &{5 6} *pp2= {5 6}
```

Working with Structs

Accessing Fields

```
// Example 08-04 Field access  
.  
  
type point struct{ x, y int }  
  
func main() {  
    p := point{2, 3}      // object  
    pp := &point{4, 5}     // pointer  
    fmt.Println("x coord of p", p.x)  
    fmt.Println("x coord of pp", pp.x)  
    pp.y = -1  
    p.y = 0  
    fmt.Println("p=", p, " pp=", *pp)  
}
```

```
[Module08]$ go run ex08-04.go  
x coord of p 2  
x coord of pp 4  
p= {2 0}  pp= {4 -1}
```

Comparing Structs

1. The operator == is defined on a struct if all the fields in the struct have the == operator defined for their type.
2. Two structs are equivalent if they are the same type and each corresponding field in the two structs is also equivalent.
3. Care has to be taken when working with pointers
4. If pp1 and pp2 are pointers to the same type of struct:
`pp1 == pp2` means "Are we both pointing to the same object?"
`*pp1 == *pp2` means "Are the objects we are pointing to equivalent?"

Struct Comparisons

```
// Example 08-05 Comparisons  
.  
type point struct{ x, y int }  
  
func main() {  
    p1 := point{2, 3}  
    p2 := point{4, 5}  
    p3 := point{2, 3}  
    fmt.Println("p1 == p2? ", p1 == p2)  
    fmt.Println("p1 == p3? ", p1 == p3)  
    pp1 := &point{1, 1}  
    pp2 := &point{1, 1}  
    fmt.Println("pp1 == pp2? ", pp1 == pp2)  
    fmt.Println("*pp1 == *pp? ", *pp1 == *pp2)  
}
```

```
[Module08]$ go run ex08-05.go  
p1 == p2? false  
p1 == p3? true  
pp1 == pp2? false  
*pp1 == *pp? true
```

Struct Pointers

1. Structs are value types just like arrays are.
2. We use slices to make passing arrays around more effective.
3. We use pointers when using structs for the same reason.
4. It is more efficient to pass pointers to structs and not the struct object.

Struct Pointers as Parameters

```
// Example 08-06 Struct pointers  
...  
type point struct{ x, y int }  
  
func swap1(p point) {  
    p.x, p.y = p.y, p.x  
    fmt.Println("After executing swap1 p=", p)  
}  
func swap2(p *point) {  
    p.x, p.y = p.y, p.x  
}  
func main() {  
    a := point{1, 2}  
    fmt.Println("Original a =", a)  
    swap1(a)  
    fmt.Println("After swap1 a =", a)  
    swap2(&a)  
    fmt.Println("After swap2 a =", a)  
}
```

```
[Module08]$ go run ex08-06.go  
Original a = {1 2}  
After executing swap1 p= {2 1}  
After swap1 a = {1 2}  
After swap2 a = {2 1}
```

Embedding Structs

Embedding Structs

```
// Example 08-07 Embedded structs 1
...
type point struct{ x, y int }

type circle struct {
    center point
    radius float32
}

func main() {
    c := circle{point{50, 32}, 13.0}
    fmt.Println("c=", c)
    c.center.x = 0
    fmt.Println("c=", c)
}
```

```
[Module08]$ go run ex08-07.go
c= {{50 32} 13}
c= {{0 32} 13}
```

Anonymous Fields

1. An anonymous field is one that only has a type but no name.
2. Fields in the inner struct act as if they are fields in the outer struct.
3. There cannot be two anonymous fields of the same type otherwise there would be no way to tell them apart.
4. Names in the outer struct shadow names in the inner struct

Anonymous Fields

```
// Example 08-08 Anonymous fields
...
type point struct{ x, y int }
type circle struct {
    point
    bool
    radius float32
}

func main() {
    c := circle{point{50, 32}, false, 3.0}
    fmt.Println("c=", c)
    c.x = 0
    c.bool = true
    fmt.Println("c=", c)
}
```

```
[Module08]$ go run ex08-08.go
c= {{50 32} false 3}
c= {{0 32} true 3}
```

Pointers as fields

```
// Example 08-09 Pointers as fields
...
type employee struct {
    fname, lname string
    id            int
    job           string
    salary        int
    boss          *employee
}

func main() {
    greta := employee{fname: "Greta", lname: "Smith"}
    anil := employee{fname: "Greta", lname: "Smith",
                     boss: &greta}
    fmt.Println("Anil's boss is", anil.boss.fname)
}
```

```
[Module08]$ go run ex08-09.go
Anil's boss is Greta
```

Pseudo-Constructors

```
// Example 08-10 Struct factory

type point struct{ x, y int }

func makePoint(x, y int) *point {
    if x < 0 {
        x = -x
    }
    if y < 0 {
        y = -y
    }
    return &point{x, y}
}

func main() {
    p1 := makePoint(1, 1)
    fmt.Println(*p1)
    p1 = makePoint(-4, -9)
    fmt.Println(*p1)
}
```

```
[Module08]$ go run ex08-10.go
{1 1}
{4 9}
```

Lab 8: Structs

Introduction to Programming in Go

9. Methods



Module Topics

1. Methods and Receivers
2. Method Sets
3. Methods on Inner Structs
4. Combining Objects

Methods and Receivers

Methods

1. Methods are functions with one modification.
2. Methods have a special "parameter" called a receiver.
3. The type of the receiver binds the method to that type.
4. Works exactly like a class method in Java.

```
func (receiver type) func_name(parameters)( return_values) {}
```

5. The binding is only indicated on the method, not in the type.
6. Methods are invoked using message type notation like Java.

Methods

```
// Example 09-01 Method for int32
...
type myint int32

func (x myint) negate() {
    x = -x
}

func main() {
    z := myint(89)
    fmt.Println(z)
    z.negate()
    fmt.Println(z)
}
```

```
[Module09]$ go run ex09-01.go
89 89
```

Pointers as Receivers

1. Like parameters, receiver variable are copies, which is why the variable "z" in the example was not negated, "x" in the method was a copy of "z".
2. Methods can take pointers as receivers.
3. If a method is applied to an object, Go automatically takes the address and uses that if the method takes a pointer receiver.
4. If the method is:
`func(x *myint) negate() {}`
then
`z.negate()` is converted to `(&z).negate()`
5. Because of the re-writing, we cannot have the same method name with a receiver of type T and a receiver of type *T.

Using a Pointer Receiver

```
// Example 09-02 Method for *int32
...
type myint int32

func (x *myint) negate() {
    *x = -*x
}

func main() {
    z := myint(89)
    fmt.Println("Before call", z)
    z.negate()
    fmt.Println("After call", z)
}
```

```
[Module09]$ go run ex08-01.go
Before call 89
After call -89
```

Method Sets

Method Sets

1. Functions cannot be overloaded in Go.
2. However functions can be partitioned into method sets.
3. Two functions are in the same method set if they have the same receiver.
4. Two methods can have the same name if they are in different method sets.
5. All of the following are ok - they are in different method sets.

```
func (x *myint) negate() {...}  
func (p * point) negate() {...}  
func negate(x float32) float32 {..}
```

Method Overloading

```
// Example 09-03 Method overloading  
...  
type myint int32  
type myfloat float64  
  
func (x *myint) negate() {  
    *x = -(*x)  
}  
func (x *myfloat) negate() {  
    *x = 0.0  
}  
  
func main() {  
    i := myint(89)  
    f := myfloat(189.9)  
    i.negate()  
    f.negate()  
    fmt.Println(i, f)  
}
```

```
[Module09]$ go run ex09-03.go  
-89 0
```

Method on Inner Structs

Inner Structs

1. If an outer struct has an inner struct as a field then:
2. The methods of the inner struct can be invoked on the outer struct.
3. But a selector has to be used.
4. This is the OO concept of aggregation and delegation.
5. The outer struct is a container for the inner structs.
6. The outer struct forwards or delegates methods to the appropriate inner struct.

Method Delegation

```
// Example 09-04 Support code

type point struct{ x, y int }

func (p *point) swap() {
    p.x, p.y = p.y, p.x
}

type circle struct {
    center point
    radius float32
}

func (c *circle) area() float32 {
    return c.radius * c.radius * math.Pi
}
```

Method Delegation

```
// Example 09-04 Inner struct methods

...

func (c *circle) area() float32 {
    c := circle{point{1, 0}, 3.0}
    fmt.Println(c)
    c.center.swap() // delegated using selector
    fmt.Println(c, c.area())
}
```

```
[ex04]$ ./ex09-04
{{1 0} 3}
{{0 1} 3} 28.274334
```

Anonymous Inner Structs

1. If the inner struct is anonymous, no selector is needed.
2. The method set of the inner struct is now part of the method set of the outer struct.
3. This emulates inheritance.
4. The inner struct acts as the parent class or super class.
5. The outer struct acts as the derived or subclass.
6. Allows for emulation of multiple inheritance.
7. Methods in the outer struct with the same name as a method in the inner struct overrides the inner struct method.

Method Incorporation

```
// Example 09-05 Changes to previous example

type circle struct {
    point      //this is now anonymous
    radius float32
}

func (c *circle) area() float32 {
    c := circle{point{1, 0}, 3.0}
    fmt.Println(c)
    c.swap() // swap is in circle's method set
    fmt.Println(c, c.area())
}
```

```
[ex05]$ ./ex09-05
{{1 0} 3}
{{0 1} 3} 28.274334
```

Overriding Methods

```
// Example 09-06 Changes to previous example

type circle struct {
    point      //this is now anonymous
    radius float32
}

func (c *circle) swap() {

}

func (c *circle) area() float32 {
    c := circle{point{1, 0}, 3.0}
    fmt.Println(c)
    c.swap() // swap is now overridden
    fmt.Println(c, c.area())
}
```

```
[ex06]$ ./ex09-06
{{1 0} 3}
{{1 0} 3} 28.274334
```

Combining Objects

Implementation of OO Inheritance

1. Aggregation: Using named inner structs.
2. Inheritance: Using anonymous inner structs.
3. Can emulate multiple inheritance.
4. Requirement that only one anonymous inner struct of a given type allows for something like virtual inheritance in C++.
5. Outer structs (subclass) methods can override inner struct (superclass) methods.

Methods and Inner Structs

Combining Methods Sets

- Given a struct with an inner struct both of which have method sets

```
type point struct{ x, y int }
type circle struct {
    center point
    radius float32
}
```

- We can apply both sets of methods by referencing the appropriate struct.

Embedded Structs 1

```
// Example 09-04 Supporting code
...
type point struct{ x, y int }

func (p *point) swap() {
    p.x, p.y = p.y, p.x
}

type circle struct {
    center point
    radius float32
}

func (c *circle) area() float32 {
    return c.radius * c.radius * math.Pi
}
```

Embedded Structs 2

```
// Example 09-04 Inner struct methods

package main

import "fmt"

func main() {

    c := circle{point{1, 0}, 3.0}
    fmt.Println(c)
    c.center.swap()
    fmt.Println(c, c.area())
}
```

```
[ex04]$ ./ex09-04
{{1 0} 3}
{{0 1} 3} 28.274334
```

Combining Objects

Combining Methods Sets

1. Changing our last example

```
type point struct{ x, y int }
type circle struct {
    point
    radius float32
}
```

2. We can apply both sets of methods on the circle struct.
3. This has the effect of the circle "inheriting" the point methods.
4. Outer struct methods can override inner struct methods.

Embedded Structs 3

```
// Example 09-05 modifications

type circle struct{
    point
    radius float32
}

...

func main() {
    c := circle{point{1, 0}, 3.0}
    fmt.Println(c)
    c.swap()
    fmt.Println(c, c.area())
}
```

```
[ex05]$ ./ex09-05
{{1 0} 3}
{{0 1} 3} 28.274334
```

Method Overloading

1. Changing our last example so that we have a swap method in both structs.
2. The cicle swap() method overrides the point swap() method.
3. To illustrate we add a swap() method to the circle that does nothing.

Method Overriding

```
// Example 09-06 Modifications to circle

type circle struct{
    point
    radius float32
}

func (c *circle) swap() {

}

func (c *circle) area() float32 {
    return c.radius * c.radius * math.Pi
}
```

Method Overriding

```
// Example 09-06 Method overloading

package main

import "fmt"

func main() {

    c := circle{point{1, 0}, 3.0}
    fmt.Println(c)
    c.swap()
    fmt.Println(c, c.area())
}
```

```
[ex06]$ ./ex09-06
{{1 0} 3}
{{1 0} 3} 28.274334
```

Lab 9: Methods

Introduction to Programming in Go

A photograph of a large-scale data center or server room. The space is filled with floor-to-ceiling server racks, all illuminated from within by a multitude of small blue lights, creating a glowing, futuristic atmosphere. The racks are arranged in long, narrow aisles that recede into the distance towards a bright overhead light fixture. The ceiling is dark and features a complex network of metal beams and structural supports.

10. Interfaces

Module Topics

1. Interfaces in Go
2. Interface Variables
3. Type Testing and Switch Statements
4. The Empty Interface

Interfaces in Go

Interfaces

1. Interfaces are defined in terms of bundles of functionality that define a "type."
2. Interfaces define a set of method signatures

```
type Swimmer interface {  
    swim()  
}
```

3. Any type which has a `swim()` method "implements" the interface and is of type "Swimmer."
4. There is no need for any declaration, just the implementation of the methods (ie. things that swim are of type `Swimmer`).

Interface Example

```
// Example 10-01 Swimmer Interface  
...  
type Swimmer interface {  
    swim()  
}  
type fish struct{ species string }  
type human struct{ name string }  
  
func (f *fish) swim() {  
    fmt.Println("Underwater")  
}  
func (f *human) swim() {  
    fmt.Println("Dog Paddle")  
}  
func (f *human) walk() {  
    fmt.Println("Strolling around")  
}
```

Interface Example

```
// Example 10-01 Swimmer Interface  
...  
func main() {  
    tuna := new(fish)  
    tuna.swim()  
    knuth := new(human)  
    knuth.new()  
    knuth.walk()  
}
```

```
[Module10]$ go run ex10-01.go  
Underwater  
Dog Paddle  
Strolling around
```

Interface Variables

Interface Variables

1. Variables can be of an interface type.
2. Any type that implements an interface can be assigned to that variable of that interface type.
3. The variable has a pointer for each method defined in the interface definition.
4. When a concrete type is assigned, the interface variable updates its pointers to reference the concrete implementations.

Interface Variables

```
// Example 10-02 Interface Variables

...
type Swimmer interface {
    swim()
}

func main() {
    var s Swimmer // s is of type "Swimmer"
    s = new(fish)
    s.swim()
    s = new(human)
    s.swim()
}
```

```
[Module10]$ ./ex10-02
Underwater
Dog Paddle
```

Interface Variable Parameters

```
// Example 10-03 Interface variables as parameters  
...  
  
type Swimmer interface {  
    swim()  
}  
  
func submerge(s Swimmer) {  
    s.swim()  
}  
  
func main() {  
    submerge(new(fish))  
    submerge(new(human))  
}
```

```
[Module10]$ ./ex10-03  
Underwater  
Dog Paddle
```

Non-interface Methods

```
// Example 10-04 Interface Variables

...
type Swimmer interface {
    swim()
}

func main() {
    var s Swimmer // s is of type "Swimmer"
    s = new(human)
    s.swim()
    s.walk() // in human but not Swimmer
}
```

```
[Module10]$ ./ex10-04
./ex10-04.go:8: s.walk undefined (type Swimmer has no field or method walk)
```

Combining Interfaces

Combining Interfaces

1. Interfaces in Go are usually small - only several methods each.
2. Interfaces can be combined like structs.
3. All of the methods are just merged into one interface.

Combining Interfaces

```
// Example 10-05 Amphib Type
...
type Swimmer interface {
    swim()
}
type Walker interface {
    walk()
}
type Amphib interface {
    Walker
    Swimmer
}

func main() {
    var a Amphib
    a = new(human)
    a.walk()
    a.swim()
}
```

```
[Module10]$ ./ex10-05
Strolling around
Dog Paddle
```

Type Testing and Switching

Type Testing

1. Example 10-04 showed that we need sometimes to know what the concrete type is of the contents of an interface variable.
2. This is done by type checking using the syntax:

`v, ok := ivar.(T)`

3. If the object in the interface variable "ivar" is of type "T" the ok is true and the object is assigned to variable v which is of type T.
4. If "ivar" is not of type "T" then ok is false and v is nil.
5. In the following example we added a few more Swimmer types and use a switch statement to do type specific processing.

Type Testing

```
// Example 10-06 Switching on types

type Swimmer interface {
    swim()
}

func main() {
    var x = []Swimmer{&human{"bobby"}, &fish{}}
    for _, swimmer := range x {
        if h, ok := swimmer.(*human); ok {
            h.walk()
        }
    }
}
```

```
[Module10]$ ./ex10-06
Strolling around
```

Type Testing

```
// Example 10-07 Switching on types

var x = []Swimmer{&human{"bobby"}, &fish{}, &cat{},
                  &squid{}}

func main() {
    for index, swimmer := range x {
        switch t := swimmer.(type) {
        case *human:
            fmt.Printf("Item %d is human and is ", index)
            t.walk()
        case *squid:
            fmt.Println("Item ", index, "is a squid")
        case *fish:
            fmt.Println("Item ", index, "is a fish")
        default:
            fmt.Printf("Item %d is type %T\n", index, t)
    }
}
```

```
[Module10]$ ./ex10-07
Item 0 is a human and is Strolling around
Item 1 is a fish
Item 2 is of type *main.cat
Item 3 is a squid
```

The Empty Interface

Empty Interface

1. The empty interface has no methods specified.
2. Everything in Go vacuously implements every empty interface type.
3. Every type, built-in or user defined in the same package as these definitions below are of type "whatever" and "stuff"

```
type whatever interface {}  
type stuff interface {}
```
4. Empty interfaces are useful for creating collections of arbitrary objects of different types.

Empty Interface

```
// Example 10-08 The empty Interface - data
...
type Swimmer interface {
    swim()
}
type myint int32
type point struct{ x, y int }

var i myint = 0

type whatever interface{}
```

Empty Interface Collection

```
// Example 10-08 The empty Interface continued -  
  
func main() {  
    mylist := []whatever{i, float64(45), &point{2, 3},  
                        true, &fish{"tuna"}}  
    for index, object := range mylist {  
        switch v := object.(type) {  
        case bool:  
            fmt.Printf("%d is bool = %v\n", index, v)  
        case myint:  
            fmt.Printf("item %d is myint = %v\n", index, v)  
        case float64:  
            fmt.Printf("item %d is a float64 = %v\n", index, v)  
        case *point:  
            fmt.Printf("item %d is a point = %v\n", index, *v)  
        default:  
            fmt.Printf("item %d is a %T = %v\n", index, v, v)  
        }  
    }  
}
```

Empty Interface Collection - Output

```
[Module10]$ ./ex10-08
item 0 is a myint = 0
item 1 is a float64 = 45
item 2 is a point = {2 3}
item 3 is a bool = true
item 4 is a *main.fish = &{tuna}
```

Lab 10: Interfaces

Introduction to Programming in Go

11. Concurrency in Go

Module Topics

1. Concurrency in Go
2. Goroutines
3. Channels
4. Select Statement
5. Race Conditions

Concurrency in Go

Concurrency in Go

1. Concurrency is not parallelism.
2. Concurrency is independently functioning computations that structure a real world task.
3. Based on Hoare's CSP where concurrency is implemented through channels where computations share information and coordinate tasks.
4. Concurrency is a fundamental part of the design of Go.

Concurrency in Go

Concurrency is implemented with four Go mechanisms

1. Concurrent function execution implemented with goroutines.
2. Synchronization and communication between goroutines (channels).
3. Multi-way concurrent control (select statement).
4. Specific Go concurrency idioms.

Goroutines

Goroutines

1. A goroutine is an independently executing function.
2. Goroutines are multiplexed into threads but goroutines are more lightweight than a thread.
3. Functions become goroutines by use of the go operator.
4. By default, the main() function is a goroutine.
5. Goroutines are very efficient resourcewise.

Normal Synchronous Call

```
// Example 11-01 Normal function call
// We have to ctrl-C to end the infinite loop

import (
    "fmt"
    "math/rand"
    "time" )

func service(message string) {
    for i := 0; ; i++ { // Infinite loop
        fmt.Println(message, i)
        time.Sleep(time.Duration(rand.Intn(1e3)) *
            time.Millisecond)
    }
}

func main() {
    service("Message:")
}
```

```
[Module11]$ go run ex11-01.go
Message: 0
Message: 1
Error: process crashed or was terminated
while running.
```

Made into Goroutine

```
// Example 11-02 Goroutine function call
// main() ends before the loop executes
...
func service(message string) {
    for i := 0; ; i++ {
        fmt.Println(message, i)
        time.Sleep(time.Duration(rand.Intn(1e3)) *
                    time.Millisecond)
    }
}

func main() {
    go service("Message:")
    fmt.Println("Done")
}
```

```
[Module11]$ go run ex11-02.go
Done!
```

Made into Goroutine

```
// Example 11-03 Goroutine function call
// goroutine executes until main() exits

func service(message string) {
    for i := 0; ; i++ {
        fmt.Println(message, i)
        time.Sleep(time.Duration(rand.Intn(1e3)) *
                    time.Millisecond)
    }
}

func main() {
    go service("Message:")
    time.Sleep(2 * time.Second)
    fmt.Println("Done")
}
```

```
[Module11]$ go run ex11-03.go
Message: 0
Message: 1
Message: 2
Message: 3
Message: 4
Message: 5
Done!
```

Multiple Goroutines

```
// Example 11-04 Goroutine function call
...
func service(message string) {
    for i := 0; ; i++ { // Infinite loop
        fmt.Println(message, i)
        time.Sleep(time.Duration(rand.Intn(1e3)) *
                    time.Millisecond)
    }
}
func main() {
    go service("Alpha:")
    go service("Beta:")
    time.Sleep(2 * time.Second)
    fmt.Println("Done")
}
```

```
[Module11]$ go run ex11-04.go
Alpha: 0
Beta: 0
Alpha: 1
Beta: 1
... snipped for space
Done!
```

Anonymous Goroutine

```
// Example 11-05 Anonymous goroutine
...

func main() {
    go func() {
        for i := 0; ; i++ {
            fmt.Println("Anon:", i)
            time.Sleep(time.Duration(rand.Intn(1e3)) *
                       time.Millisecond)
        }
    }()
    time.Sleep(2 * time.Second)
    fmt.Println("Done")
}
```

```
[Module11]$ go run ex11-05.go
Anon: 0
Anon: 1
Anon: 2
Anon: 3
Anon: 4
Anon: 5
Done!
```

Channels

Channels

1. Channels are modeled after a socket or a pipe in Unix.
2. Channels are first class objects, like functions.
3. Channels are bidirectional and goroutines communicate by reading from and writing to channels.
4. Channels have types and are created with make(). In the code below "c" is a channel that passes ints.

`c := make(chan int)`

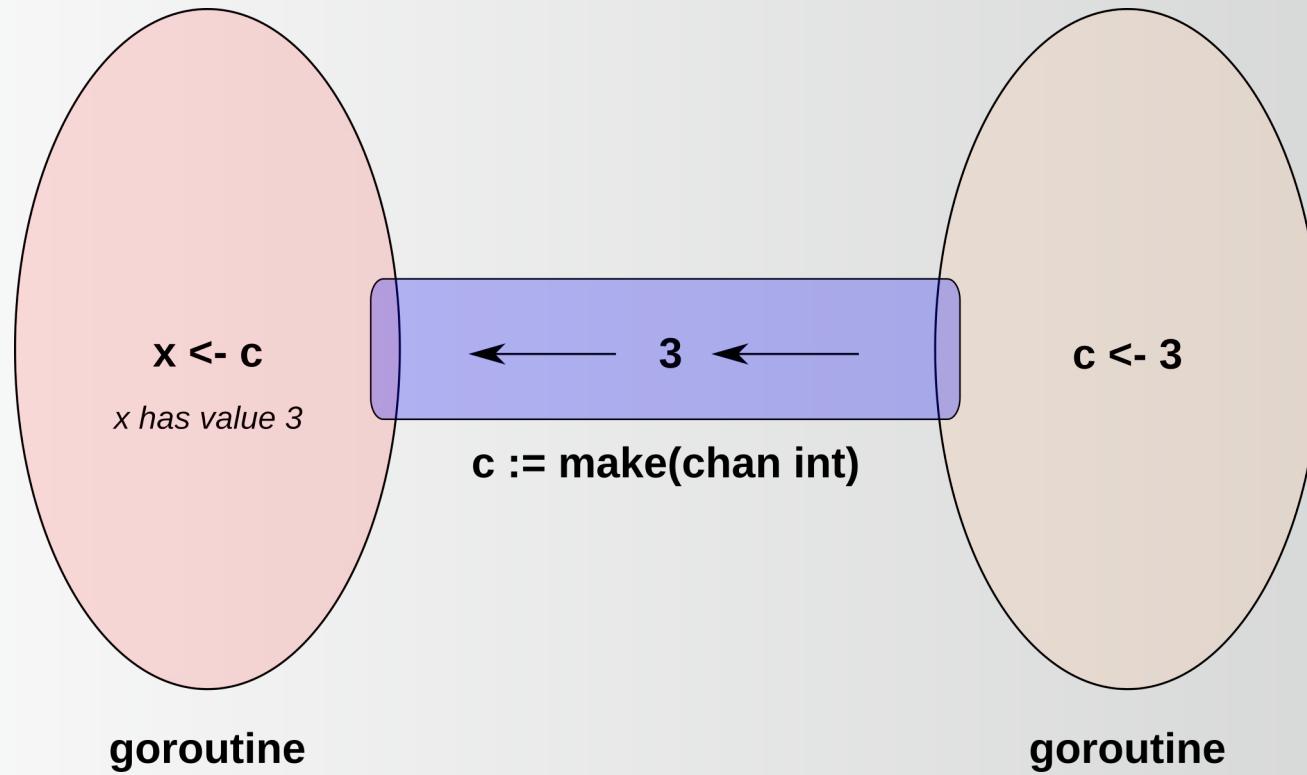
5. To write to the channel c we use the syntax

`c <- 2`

and to read from a channel c we use the syntax

`myvariable <- c`

Communicating with Channels



Communicating with Channels

1. The value 3 is written onto the channel by the goroutine on the right.
2. The goroutine on the left reads it from the channel and assigns it to the variable x.
3. Channels will block on either side if the channel transmission cannot be completed.
4. The goroutine on the left blocks until there is something to read.
5. The goroutine on the right blocks until someone reads from the channel it just wrote to.
6. The channel "synchronizes" the activity of the two goroutines.

Basic Channel

```
// Example 11-06 Basic Channel
...
func service(message string, c chan string) {
    for i := 0; ; i++ {
        fmt.Println(message, i)
        c <- fmt.Sprintf("%s %d", message, i)
    }
}
func main() {
    c := make(chan string)
    go service("Message:", c)
    for i := 0; i < 5; i++ {
        fmt.Printf("main %d Got back: %q\n", i, <-c)
        time.Sleep(time.Second)
    }
    fmt.Println("Done")
}
```

Output for Example 11-06

```
[Module11]$ go run ex11-06.go
service: 0
service: 1
main 0 Got back: "Message: 0"
main 1 Got back: "Message: 1"
service: 2
main 2 Got back: "Message: 2"
service: 3
main 3 Got back: "Message: 3"
service: 4
main 4 Got back: "Message: 4"
service: 5
Done!
```

Deadlocks

1. Because channels block, we can deadlock.
2. Deadlocks usually occur when both sides are waiting for the other goroutine participating in a channel to do something.
3. Go detects deadlocks and issues a panic when one occurs.

Deadlocks

```
// Example 11-07 Deadlocks
// Both goroutines are waiting to read

func service(message string, c chan string) {
    c <- "Hello"    //write
    fmt.Println(<-c) //read
}

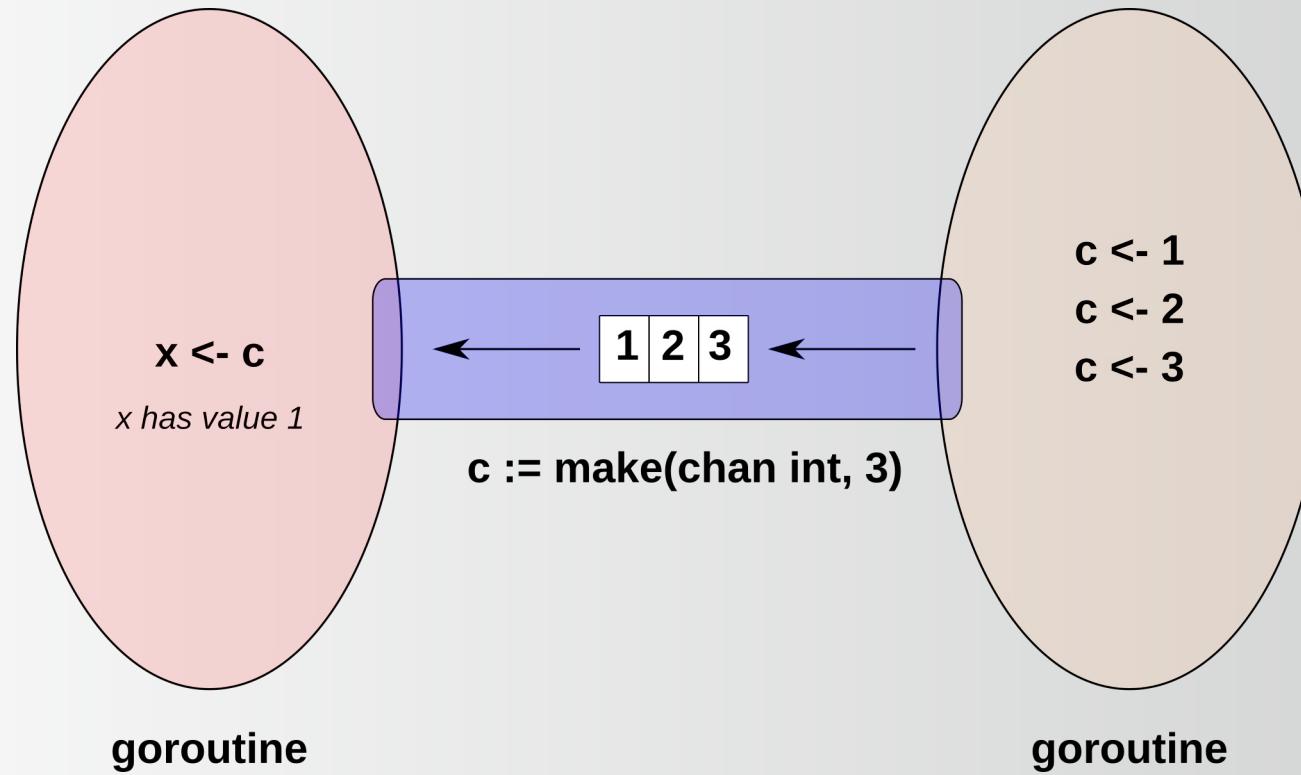
func main() {
    c := make(chan string)
    go service("Message:", c)
    c <- "Bye"     //write
    fmt.Println(<-c) //read
}
```

```
[Module11]$ go run ex11-07.go
fatal error: all goroutines are asleep - deadlock!
```

Buffered Channels

1. Buffers can be attached to channels.
2. Useful when there is a mismatch between the rate of reading and writing between the two goroutines.
3. Buffered channels are made with an initial capacity:
`c := make(chan int, 5)`
4. When the buffer is full, the channel blocks just like an unbuffered channel.
5. Buffers make channels less synchronized which can introduce difficulties when coordinating goroutines.
6. The capacity of the buffer is `cap(buffer)` and the number of items in the buffer is `len(buffer)`

Buffered Channels



Communicating with Channels

1. The values 1,2,3 are written onto the channel by the goroutine on the right..
2. The goroutine on the right blocks since the buffer is full.
3. The goroutine on the left reads 1 from the buffer.
4. The goroutine on the right can now write a new value.

Basic Buffered Channel

```
// Example 11-08 Buffered Channels
...
func service(message string, c chan string) {
    for i := 0; ; i++ {
        fmt.Println(message, i, "buffered items ", len(c))
        c <- fmt.Sprintf("%s %d", message, i)
    }
}
func main() {
    c := make(chan string, 3)
    go service("Message:", c)
    for i := 0; i < 5; i++ {
        fmt.Printf("main %d Got back: %q\n", i, <-c)
        time.Sleep(time.Second)
    }
    fmt.Println("Done")
}
```

Output for Example 11-08

```
[Module11]$ go run ex11-08.go
service: 0 buffered items 0
service: 1 buffered items 0
service: 2 buffered items 1
service: 3 buffered items 2
service: 4 buffered items 3
main 0 Got back: "Message: 0"
main 1 Got back: "Message: 1"
service: 5 buffered items 3
main 2 Got back: "Message: 2"
service: 6 buffered items 3
service: 7 buffered items 3
main 3 Got back: "Message: 3"
main 4 Got back: "Message: 4"
service: 8 buffered items 3
Done!
```

Unidirectional Channels

1. Channels are bidirectional because someone has to write to it and someone has to read from it.
2. When a channel is passed to a goroutine it may be specified as an input only or output only channel.
3. Specifying a direction only specifies how to use the channel inside that routine - it is not property of the channel itself.
4. Trying to read on an output channel or write on an input channel is an error.
5. Direction is indicated like this

`chan<-` (output)

`<-chan` (input)

Unidirectional Channel

```
// Example 11-09 Unidirectional Channel

func service(message string, c chan<- string) {
    for i := 0; ; i++ {
        fmt.Println(message, i)
        c <- fmt.Sprintf("%s %d", message, i)
    }
}

func main() {
    c := make(chan string)
    go service("Message:", c)
    for i := 0; i < 5; i++ {
        fmt.Printf("main %d Got back: %q\n", i, <-c)
        time.Sleep(time.Second)
    }
    fmt.Println("Done")
}
```

Closing Channels

1. A close() operation on a channel is performed by a sender to indicate that no more data will be sent.
2. Trying to write to a closed channel causes a panic.
3. It is an error to try and close a read only channel.

Closing a Channel

```
// Example 11-10 Closing a Channel
...
func service(message string, c chan<- string) {
    for i := 0; ; i++ {
        fmt.Println(message, i)
        c <- fmt.Sprintf("%s %d", message, i)
        close(c) // will cause a panic next iteration
    }
}
func main() {
    c := make(chan string)
    go service("Message:", c)
    for i := 0; i < 5; i++ {
        fmt.Printf("main %d Got back: %q\n", i, <-c)
        time.Sleep(time.Second)
    }
    fmt.Println("Done")
}
```

```
[Module11]$ go run ex11-10.go
service: 0
service: 1
panic: send on closed channel
```

Closing Channels as a Signal

1. The range function will read on a channel until it is closed.
2. When the sender is finished sending data they close the channel.
3. Closing a channel causes the range function reading from it to terminate.
4. This ensures that the receiver is not blocked waiting for input that will never come.

Closing a Channel as a Signal

```
// Example 11-11 Closing a Channel
...
func sender(output chan<- int) {
    for i := 0; i < 3; i++ {
        output <- i
    }
    close(output)
}
func receiver(input <-chan int) {
    for j := range input {
        fmt.Println("Received ", j)
    }
    fmt.Println("I'm done ")
}
func main() {
    c := make(chan int)
    go receiver(c)
    go sender(c)
    time.Sleep(time.Second)
}
```

```
[Module11]$ go run ex11-11.go
Received 0
Received 1
Received 2
I'm done
```

The Blocking Problem

1. The previous example required a `close()` on the channel to prevent blocking.
2. However it may happen that the sender exits before the channel is closed, then the receiver blocks.
3. A Go idiom is to start a deferred goroutine that closes the channel whenever the sender exits.
4. We can check explicitly to see if a channel is closed by using the comma ok idiom.

Using comma ok

```
// Example 11-12 Go idioms
...
func sender(output chan<- int) {
    defer close(output) // ensures no blocking
    for i := 0; i < 3; i++ {
        output <- i
    }
}
func receiver(input <-chan int) {
    for {
        j, ok := <-input
        fmt.Println(ok)
        if !ok {
            break
        }
        fmt.Println("Received ", j)
    }
    fmt.Println("I'm done ")
}
func main() {} //as before
```

Output for Example 11-12

```
[Module11]$ go run ex11-12.go
true
Received 0
true
Received 1
true
Received 2
false
I'm done
```

The Select Statement

The Select Statement

1. The select statement is a channel polling mechanism that works like a switch statement.
2. Each case in the select statement is an input channel.
3. Each iteration, input is read from a channel that is not blocked.
4. If more than one channel is not blocked, then one of the unblocked channels is chosen at random.
5. If all the channels are blocked, then the default case is executed.

Select Statement

```
// Example 11-13 Generator goroutines

...

func sourceEven(outchan chan<- int) {
    for i := 0; ; i++ {
        output <- i * 2
        time.Sleep(time.Duration(rand.Intn(1e3)) *
                    time.Millisecond)
    }
}

func sourceOdd(outchan chan<- int) {
    for i := 0; ; i++ {
        output <- (i * 2) + 1
        time.Sleep(time.Duration(rand.Intn(1e3)) *
                    time.Millisecond)
    }
}
```

Select Statement

```
// Example 11-13 Multiplexer goroutine
...
func counter(ceven <-chan int, codd <-chan int) {
    for {
        select {
        case e := <-ceven:
            fmt.Println("Even:", e)
        case o := <-codd:
            fmt.Println("Odd:", o)
        default:
            fmt.Println("no one is ready")
            time.Sleep(100 * time.Millisecond)
        }
    }
}
```

Select Statement

```
// Example 11-13 Main goroutine
...
func main() {
    codd := make(chan int)
    ceven := make(chan int)
    go sourceOdd(codd)
    go sourceEven(ceven)
    go counter(ceven, codd)
    time.Sleep(2 * time.Second)
}
```

```
[Module11]$ go run ex11-13.go
Even: 0
Odd: 1
no one is ready
Odd: 3
no one is ready
no one is ready
no one is ready
no one is ready
Even: 2
no one is ready
Odd: 5
Even: 4
no one is ready
Even: 6
no one is ready
no one is ready
no one is ready
Odd: 7
no one is ready
no one is ready
Even: 8
no one is ready
```

Race Conditions

Race Conditions

1. Channels are concurrency safe.
2. Often concurrency requires two goroutines to access the same variable or location in memory.
3. Race conditions mean interleaved reads and writes to the shared variable create some form of value corruption.
4. Race conditions usually only occur when the system is in a certain state, loading or stress.
5. Race conditions are very difficult to detect in testing.

Go Solution to Race Conditions

1. A Go idiom to prevent race conditions is to have the shared variable only accessible by one goroutine.
2. Requests to read and write are send to that goroutine over channels.
3. This generally will prevent the sort of environment necessary for a race condition.
4. The net effect is to "wrap" the resource in a goroutine and allow access only via channels which are concurrency safe.

Go Locking

1. If the Go Idiom is not feasible, then the resource can be locked with a mutex.
2. Each go routine that needs access locks the variable, accesses it then unlocks it.
3. This is a more traditional approach in other programming languages.

Lab 11: Concurrency

Introduction to Programming in Go

A photograph of a large-scale data center or server room. The space is filled with floor-to-ceiling server racks, all illuminated from within by a multitude of small blue and white lights. The perspective is looking down a central aisle between two long rows of these racks, which recede into the distance towards a bright light at the far end of the room.

12. Testing in Go

Module Topics

1. Testing in Go
2. Unit Testing in Go
3. Benchmarking in Go
4. Profiling in Go

Testing in Go

Go Testing

1. The go test tool performs three kinds of testing:
2. i) Unit testing using test functions.
3. ii) Benchmarking using benchmark functions.
4. iii) Examples, which we will not do in the course.
5. The go test tool works like go build does but does an alternative kind of build that is test oriented.
6. This is an attempt to reduce dependencies on third party tools.

Unit Testing in Go

Go Testing

1. For each file that contains functions we want to test we create a corresponding test file with addition of "_test" at the end of the file name.
2. For example, the test file for "counter.go" is "counter_test.go"
3. Go build/install tools ignore the test files, Go test does not.
4. The test functions must all be of the form

```
func Testxxx(t *testing.T)
```
5. The xxx can be whatever you want.
6. The testing.T struct accesses the test framework and logging structs to record the test results.
7. The "testing" package must be imported.

A Function to Test

```
// Example 12-01 Buggy Program to Test

func count(s string) (vowels, consonants int) {
    for _, letter := range s {
        switch letter {
        case 'a', 'e', 'i', 'o', 'u':
            vowels++
        default:
            consonants++
        }
    }
    return
}
func main() {
    input := "This is a test"
    v, c := count(input)
    fmt.Printf("vowels=%d, consonants=%d\n", v, c)
}
```

```
[Module12]$ go run ex12-01.go
vowels=4, consonants=10
```

The Test Functions

```
// Example 12-01 Unit Tests
import "testing"

func TestOne(t *testing.T) {
    v, c := count("a test case")
    if c != 5 {
        t.Error("Test One: Expected 5 cons, got ", c)
    }
    if v != 4 {
        t.Error("Test One: Expected 4 vowels, got ", v)
    }
}
func TestTwo(t *testing.T) {
    v, c := count("And more stuff")
    if c != 8 {
        t.Error("Test Two: Expected 8 cons, got ", c)
    }
    if v != 4 {
        t.Error("Test Two: Expected 4 vowels, got ", v)
    }
}
```

Running the Tests

```
[Module12-01]$ go test
--- FAIL: TestOne (0.00s)
    ex12-01_test.go:9: Test One: Expected 5 cons, got  7
--- FAIL: TestTwo (0.00s)
    ex12-01_test.go:18: Test Two: Expected 8 cons, got  11
    ex12-01_test.go:21: Test Two: Expected 4 vowels, got  3
FAIL
exit status 1
FAIL    examples/Module12/ex12-01 0.002s
```

A Fixed Function to Test

```
// Example 12-02 Buggy Program to Test Fixed
...
func count(s string) (vowels, cons int) {
    for _, letter := range s {
        switch letter {
        case 'a', 'e', 'i', 'o', 'u':
            vowels++
        case 'A', 'E', 'I', 'O', 'U':
            vowels++
        case ' ':
            break
        default:
            cons++
        }
    }
    return
}
```

```
[Module12-02]$ go test
PASS
ok      examples/Module12/ex12-02 0.002s
```

Coverage

1. Coverage is a measure of how many statements were executed by the tests.
2. Coverage as measured by the tool is not a precise metric.
3. Low coverage values often mean that you are not testing the code you think you are testing.
4. Coverage should always be considered an estimate.
5. The next slide shows coverage measures for the examples presented so far

Coverage Measures

```
Module12-01]$ go test -cover
--- FAIL: TestOne (0.00s)
    ex12-01_test.go:9: Test One: Expected 5 cons, got 7
--- FAIL: TestTwo (0.00s)
    ex12-01_test.go:18: Test Two: Expected 8 cons, got 11
    ex12-01_test.go:21: Test Two: Expected 4 vowels, got 3
FAIL
coverage: 62.5% of statements
exit status 1
FAIL      examples/Module12/ex12-01 0.002s
```

```
[Module12-02]$ go test -cover
PASS
coverage: 70.0% of statements
ok      examples/Module12/ex12-02 0.002s
```

Benchmarking

Benchmarking

1. Benchmarking is measuring the performance of a function under a fixed workload.
2. Benchmarking functions have the forms:

```
func BenchmarkXxx(b *testing.B)
```

and they are in the same file as the unit tests.

3. testing.B is a struct that is used to track benchmarking information.
4. By default no benchmarking functions are run by “go test,” they have to be specified on the command line with a regular expression that matches the names of the benchmark functions to be run.
5. Benchmarks are run in a for loop where the benchmarking program runs the function to be measured b.N times with increasingly large values of N until the benchmark values stabilize.

Function to Benchmark

```
// Example 12-03 Program to be Benchmarked
...
func Fibonacci(n int) int {
    if n < 2 {
        return n
    }
    return Fibonacci(n-1) + Fibonacci(n-2)
}

func main() {
    fmt.Println(Fibonacci(30))
}
```

```
[Module12-03]$ go run ex12-03.go
832040
```

The Benchmark Function

```
// Example 12-03 Benchmarking

import "testing"

func BenchmarkFib20(b *testing.B) {
    for n := 0; n < b.N; n++ {
        Fibonacci(20)
    }
}
```

```
[Module12-03]$ go test -bench=.
testing: warning: no tests to run
PASS
BenchmarkFib20-12            30000          40771 ns/op
ok      _examples/Module12/ex12-03 1.728s
```

Benchmarking

1. The middle number is the final value of b.N, the test ran for 30,000 loops with an average time of 40771 nanoseconds per loop.
2. A complete breakdown of benchmarking in general is beyond this introduction.

Profiling

Profiling

1. Measures the performance of critical code.
2. Samples a number of events during the program execution and then extrapolates to produce a statistical summary called a profile.
3. Three different kinds of profiling are supported by Go testing.

CPU profiling - functions whose execution requires the most CPU time.

Heap profiling - statements responsible for allocating the most memory.

Blocking profiling - operations responsible for blocking goroutines the longest.

4. A profile is collected in a log file which can then be examined using the pprof Go tool.
5. No more than one profile at a time should be done or the sampling will be skewed.

Profiles

```
[Module12-04]$ go test -bench=. -cpuprofile=cout.log
testing: warning: no tests to run
PASS
BenchmarkFib20-12           30000          40828 ns/op
ok      _/examples/Module12/ex12-04 1.671s
```

```
[Module12-04]$ go tool pprof -text ex12-04.test cout.log
1.67s of 1.67s total ( 100%)
      flat  flat%  sum%          cum   cum%
      1.65s 98.80% 98.80%    1.65s 98.80%  Module12/ex12-04.Fibonacci
      0.01s  0.6% 99.40%    0.01s   0.6%  runtime.(*mspan).sweep
      0.01s  0.6% 100%     0.01s   0.6%  runtime.usleep
      0     0% 100%     1.65s 98.80%  Module12/ex12-04.BenchmarkFib20
      0     0% 100%     0.01s   0.6%  runtime.(*gcWork).get
      0     0% 100%     0.01s   0.6%  runtime.GC
      0     0% 100%     0.01s   0.6%  runtime.findRunnable
      0     0% 100%     0.01s   0.6%  runtime.gcDrain
      0     0% 100%     0.01s   0.6%  runtime.gcMarkTermination
      0     0% 100%     0.01s   0.6%  runtime.gcMarkTermination.func2
      0     0% 100%     0.01s   0.6%  runtime.gcStart
      0     0% 100%     0.01s   0.6%  runtime.gcSweep
      0     0% 100%     0.01s   0.6%  runtime.gcHelper
      0     0% 100%     0.01s   0.6%  runtime.getfull
      0     0% 100%     1.66s 99.40%  runtime.goexit
      0     0% 100%     0.01s   0.6%  runtime.mstart
      0     0% 100%     0.01s   0.6%  runtime.mstart1 ...
```

1. The way of profiling demonstrated so far is cumbersome.
2. Dave Cheney has written a nice interface to wrap around profiling code.
3. The utility is available from a github repository.
4. Its use is demonstrated in the example.

Function to Profile

```
// Profiling non-test functions
...
import (
    "fmt"
    "github.com/pkg/profile"
)
func Fibonacci(n int) int {
    if n < 2 {
        return n
    }
    return Fibonacci(n-1) + Fibonacci(n-2)
}

func main() {
    defer profile.Start().Stop()
    fmt.Println(Fibonacci(30))
}
```

Output

```
[Module12-04]$ go run ex12-04.go
2016/09/22 07:34:59 profile: cpu profiling enabled, /tmp/profile457278947/
cpu.pprof
Fibonacci num for 30 is 1346269
```

```
[Module12-04]$ go tool pprof -text ex12-04 /tmp/profile457278947/cpu.pprof
10ms of 10ms total ( 100%)
      flat  flat%   sum%      cum  cum%
      10ms  100%  100%      10ms  100%  main.fib
          0    0%  100%      10ms  100%  main.main
          0    0%  100%      10ms  100%  runtime.goexit
          0    0%  100%      10ms  100%  runtime.main
```

Lab 12: Testing

Introduction to Programming in Go

A photograph of a large-scale data center or server room. The perspective is looking down a central aisle between two long rows of server racks. The racks are densely packed with hardware, and numerous blue and yellow lights are visible, glowing from the front panels and internal components of the servers. The ceiling above is dark and features a complex network of overhead infrastructure, including pipes and beams. The floor is a polished concrete surface.

13. Go Packages

Module Topics

1. Packages
2. Workspace Organization
3. Initialization
4. Using `go get`
5. Vendorizing

Packages

Go Packages

1. Packages are located by their import path.
2. Location information is never "inside" a package file, packages files are located exclusively by the go tool.
3. The last part of a package import path is the local name for the package and used as a prefix .
4. Custom packages should use a naming protocol like that for XML namespaces to ensure unique names. For example
`import "capitalone.com/devteam6/roberto/utilities"`
5. Public symbols are capitalized, all others are private to the package.

Workspace Organization

The "Go" Way

1. All of a programmers's Go code is in a single workspace.
2. A workspace contains many version controlled repositories.
3. Each repository contains one or more packages.
4. A package has one or more Go source files in a single directory.
5. The path to a package's directory determines its import path.

Workspace Organization

1. A workspace is a directory hierarchy with three directories at its root:
 - `src`: contains all of the Go source files,
 - `pkg`: contains package objects, and
 - `bin`: contains executable commands.
2. The `pkg` objects are compiled files.
3. The `pkg` directory contains subdirectories based on the compilation target architecture.

Adding a pkg to "Hello world"

```
// Example 13-01 Stringutils package

package stringutils

func Reverse(s string) string {
    r := []rune(s)
    for i,j := 0, len(r)-1; i < len(r)/2; i,j = i+1, j-1
    {
        r[i], r[j] = r[j], r[i]
    }
    return string(r)
}
```

Using the *stringutil* package

```
// Example 13-01 Hello.go

package main

import (
    "fmt"
    "stringutil"
)

func main() {
    fmt.Println(stringutil.Reverse("Hello World!"))
}
```

The Project Structure - Build

```
src\  
  hello\  
    hello.go  
  stringutil\  
    stringutil.go  
pkg\  
bin\
```

```
[Module13]$ cd $GOPATH/src/hello  
[stringutil] go build hello  
[stringutil] ls  
hello.go hello
```

The Project Structure - Install

```
src\  
  hello\  
    hello.go  
  stringutil\  
    stringutil.go  
pkg\  
  linux_amd64\  
    stringutil.a  
bin\  
  hello
```

```
[Module13]$ cd $GOPATH/src/hello  
[stringutil] go install hello  
[stringutil] ls  
hello.go hello
```

Package Aliases

1. Package prefixes are the last part of the import path.
2. This can lead to ambiguities.
3. We can define local aliases for the package prefixes.
4. Consider adding a new `stringutils` package with a different import path but identical local prefix to the Hello World example.
5. The new package has a different version of the `Reverse()` method which also converts a reversed string to uppercase.

Adding another pkg to "Hello world"

```
// Example 13-03 utils/stringutils package

package stringutils
import "strings"

func Reverse(s string) string {
    r := []rune(s)
    for i,j := 0, len(r)-1; i < len(r)/2; i,j = i+1, j-1
    {
        r[i], r[j] = r[j], r[i]
    }
    return strings.Toupper(string(r))
}
```

Using Both the `stringutil` Packages

```
// Example 13-04 Hello.go with ambiguities

package main

import (
    "fmt"
    "stringutil"
    "util/stringutil"
)

func main() {
    fmt.Println(stringutil.Reverse("Hello World!"))
    fmt.Println(stringutil.Reverse("Hello World!"))
}
```

```
[hello]$ go build hello
./hello.go:7: stringutil redeclared as imported package name
        previous declaration at ./hello.go:6
Error: process exited with code 2.
```

Using Both the *stringutil* Packages

```
// Example 13-05 Hello.go with local aliases

package main

import (
    "fmt"
    s1 "stringutil"
    s2 "util/stringutil"
)

func main() {
    fmt.Println(s1.Reverse("Hello World!"))
    fmt.Println(s2.Reverse("Hello World!"))
}
```

```
[hello]$ go build hello
./hello
!dlrow olleH
!DLROW OLLEH
```

Package Initialization

Package Initialization

1. In each file there can be one or more functions named init().
2. All init() functions are executed after all of the package variable have been initialized; and
3. All of the init() functions from the imported packages have run.
4. The purpose of the init() functions verify and repair program state before execution.

Using Both the `StringUtil` Packages

```
// Example 13-06 Hello.go with init()

import (
    "fmt"
    s1 "StringUtil"
    s2 "util/StringUtil"
)
func init() {
    fmt.Println("Main init 1")
}
func init() {
    fmt.Println("Main init 2")
}
func main() {
    fmt.Println(s1.Reverse("Hello World!"))
    fmt.Println(s2.Reverse("Hello World!"))
}
```

```
[hello]$ go build hello
./hello
Main init 1
Main init 2
!dlrow olleH
!DLROW OLLEH
```

Init() in a Dependency

```
// Example 13-08 utils/stringutils package init()

package stringutils
import "strings"
import "fmt"

func init() {
    fmt.Println("util/stringutil init")
}

func Reverse(s string) string {
    r := []rune(s)
    for i, j := 0, len(r)-1; i < len(r)/2; i, j = i+1, j-1
    {
        r[i], r[j] = r[j], r[i]
    }
    return strings.ToUpper(string(r))
}
```

Init() in a Dependency

```
// Example 13-07 Hello.go with init()

import (
    "fmt"
    s1 "stringutil"
    s2 "util/stringutil"
)
func init() {
    fmt.Println("Main init 1")
}
func init() {
    fmt.Println("Main init 2")
}
func main() {
    fmt.Println(s1.Reverse("Hello World!"))
    fmt.Println(s2.Reverse("Hello World!"))
}
```

```
[hello]$ go build hello
./hello
util/stringutil init
Main init 1
Main init 2
!dlrow olleH
!DLROW OLLEH
```

Blank Alias

1. If we want to force a dependency init() to execute but we don't want to import any symbols we use "_" for the local alias.
2. This disables the compiler from generating an error that we have a package import statement and no symbols imported from that package.

Blank Alias

```
// Example 13-08 Blank Alias

import (
    "fmt"
    "stringutil"
    _ "util/stringutil"
)
func init() {
    fmt.Println("Main init 1")
}
func init() {
    fmt.Println("Main init 2")
}
func main() {
    fmt.Println(stringutil.Reverse("Hello World!"))
}
```

```
[hello]$ go build hello
./hello
util/stringutil init
Main init 1
Main init 2
!dlrow olleH
```

Go Get

Getting Remote Packages

1. Allows Go to get packages from remote locations specific by a URL
2. Works with most repository and version control systems and has a rich range of options.
3. Copies the source code to the current workspace src directory, then does the equivalent of a go install on the downloaded source.
4. In the example, we get a utility called golint from a github repository starting with an empty workspace

Workspace and go get Command

```
src\  
pkg\  
bin\
```

```
[Module14]$ go get github.com/golang/lint/golint
```

```
bin\  
  golint  
pkg\  
  linux_amd64\  
    github.com\  
      golang\  
        lint.a  
golang.org\  
  x\  
    tools\  
      go\  
        gcimporter15.a
```

Workspace and go get Command

```
src\  
  github.com\  
    golang\  
      lint\  
  golang.org\  
    x\
```

Vendor Management

Vendoring

1. The drawback to the Go workspace organization is that there is only one version of a package available.
2. Problem 1: Two developers in the same workspace are using different versions of a package.
3. Problem 2: Two different projects require different versions of a package.
4. Go is designed to work with local copies of remote packages so that changes or missing remote repositories do not break local builds.
5. This is called "vendoring."

Vendoring

1. If two different versions of a package are needed, the import paths can be rewritten so that they are both installed but with different paths.
2. Not an effective solution since it involves a lot of low level work.
3. Too easy to make errors manually that will break a build.

Vendoring Third Party Tools

1. Third party tools like "godep" can be used to take snapshots of versions used in a build.
2. For example using "godep save" creates a json file of version information.
3. Then using "godep go build" feeds the correct version information into the build utility.
4. The use of third party tools though is not what was intended for Go.

A godep Snapshot

```
"ImportPath": "github.com/golang/prog",
  "GoVersion": "go1.6",
  "Deps": [
    {
      "ImportPath": "github.com/golang/examples/ex1",
      "Rev": "e0e1b550d545d9be0446ce324babcb16f09270f5"
    },
    {
      "ImportPath": "ithub.com/golang/examples/ex2",
      "Rev": "a1577bd3870218dc30725a7cf4655e9917e3751b"
    },
  )
}
```

The Vendor Folder

1. The latest version of Go has introduced the idea of a vendor folder.
2. If there is a vendor folder in your src tree, then anything under the folder is an external vendor dependency.
3. When building the project, the go tools will check under the vendor folder first for the package rather than look outside your project.
4. Two different versions of a package can be kept in different projects by placing them in different vendor folders in each project.
5. Most of the third party vendorizing tools support this feature.

The Vendor Folder

```
src\  
  hello\  
    vendor\  
      stringutil\  
        version 1 code  
  bye\  
    vendor\  
      stringutil\  
        version 2 code  
  stringutil\  
    version3 code  
  whatever\  
  pkg\  
  bin
```

The Vendor Folder

1. The hello project will use version 1 of stringutil package.
2. The bye project will use version 2 of stringutil package.
3. The whatever project, and any other projects, will use version 3 of stringutil package

Lab 13: Packages

Introduction to Programming in Go

A photograph of a large-scale data center or server room. The perspective is looking down a central aisle between two long rows of server racks. The racks are densely packed with hardware, and numerous blue and yellow lights are visible, glowing from the front panels and internal components of the servers. The ceiling above is dark with complex metal trusses and lighting fixtures. The floor is a polished concrete surface.

14. Strings, Patterns and Files

Module Topics

1. Characters, Unicode and Runes
2. Regular Expressions and Patterns
3. String Conversions
4. File Input and Output
5. Vendoring

Characters, Unicode and Runes

Go Strings

1. Strings are immutable and implemented as arrays of bytes.
2. The content of the individual bytes is arbitrary, no restrictions.
3. Treating a string as a slice of bytes makes transmission over different media easier since it is just binary data .
4. The length of a string returned by `len()` is the number of bytes not the number of characters.
5. Characters are Unicode code points and require 4 bytes to encode.
6. Go encodes code points in UTF-8 so characters have variable width from 1 to 4 bytes.

Example 14-01

```
1 // Example 14-01 Basic Strings
2
3 package main
4
5 import "fmt"
6
7 func main() {
8     const junk = "\xbd\xb2\x3d\xbc\x20\xe2\x8c\x98"
9     const french = "Côté"
10    const ascii = "London"
11
12    fmt.Printf("junk=%s| french=%s| ascii=%s|\n", junk, french, ascii)
13    fmt.Printf("Lengths junk=%d french=%d ascii=%d\n", len(junk), len(french), len(ascii))
14 }
15
```

Build Output

```
/usr/local/go/bin/go run ex14-01.go [/home/gospace/Module14]
junk =|\u20bd\u20b2\u203d\u20bc\u2020\u00e2\u208c\u2098| french=|Côté| ascii=|London|
Lengths junk=8 french=6 ascii=6
Success: process exited with code 0.
```

String Literals

1. Two forms of string literals.
2. Interpreted strings in double quotes allow the use of escaped character sequences like \n or \xa2.
3. Raw strings in back quotes (not single quotes) does not interpret escaped character sequences.
4. The escape sequence \Unnnn is used for Unicode code points.

Example 14-02

```
1 // Example 14-02 String Literals
2
3 package main
4
5 import "fmt"
6
7 func main() {
8
9     const s1 string = "日本語"                      // UTF-8 input text
10    const s2 string = `日本語`                        // UTF-8 input text as a raw literal
11    const s3 string = "\u00e5\u00d7\u00a9"              // the explicit Unicode code points
12    const s4 string = "\u0000e5\u0000d7\u0000a9"        // the explicit Unicode code points
13    const s5 string = "\xe6\x97\xab\xac\xea\x9c\x9e" // the explicit UTF-8 bytes
14    fmt.Printf("s1=%s| s2=%s| s3=%s| s4=%s| s5=%s|\n", s1, s2, s3, s4, s5)
15    fmt.Println("Length of string: ", len(s1))
16 }
17
```

Build Output ▾ ■ 🔍 ⚙

```
/usr/local/go/bin/go run ex14-02.go [/home/gospace/Module14]
s1=|日本語| s2=|日本語| s3=|日本語| s4=|日本語| s5=|日本語|
Length of string: 9
Success: process exited with code 0.
```

Iterating over Strings

1. Using a for loop iterates byte by byte.
2. For loops are used when we want to iterate through a string as a sequence of binary bytes.
3. The range function iterates character by character.
4. The range function interprets 1-4 bytes per iteration as a character.

Example 14-03

```
1 // Example 14-03 Iteration over strings
2
3 package main
4
5 import "fmt"
6
7 func main() {
8
9     const french string = "Côté"
10    fmt.Println("Iterating with for loop")
11    for i := 0; i < len(french); i++ {
12        fmt.Printf("%x ", french[i])
13    }
14    fmt.Printf("\nIterating with range (%d bytes)\n", len(french))
15    for index, runeValue := range french {
16        fmt.Printf("%#U starts at byte position %d with length\n", runeValue, index)
17    }
18 }
19
```

Build Output ▾ ■ 🔍 ⚙

```
/usr/local/go/bin/go run ex14-03.go [/home/gospace/Module14]
Iterating with for loop
43 c3 b4 74 c3 a9
Iterating with range (6 bytes)
U+0043 'C' starts at byte position 0 with length
U+00F4 'ô' starts at byte position 1 with length
U+0074 't' starts at byte position 3 with length
U+00E9 'é' starts at byte position 4 with length
Success: process exited with code 0.
```

Unicode

1. International standard to allow representation of text in all writing systems and orthographies.
2. Requires 4 bytes per symbol - called code points and not characters.
3. The first 256 Unicode values are ASCII so all ASCII strings are also Unicode strings.
4. Also includes archaic writing forms like Egyptian Hieroglyphics as well as scientific symbols and emojis.

UTF-8

1. Developed by Rob Pike and Ken Thompson who also developed Go.
2. Uses between 1-4 bytes to encode 32-bit code points.
3. High order bit on first byte is 0 if code point is 1 byte
4. If the high order bit is 1, then the next three bits encode the number of bytes total ($110 = 2$ bytes, $1110 = 3$ bytes, $11110 = 4$ bytes)
5. The extra bytes for the code point all start with 10....
6. Since most character based scripts need only 2 bytes and ASCII only needs 1 byte per character, allows for very efficient "packing".

UTF-8 Encoding

Character Length	First Byte	Second Byte	Third Byte	Fourth Byte
1 Byte	0xxxxxx			
2 Bytes	110xxxxx	10xxxxxx		
3 Bytes	1110xxxx	10xxxxxx	10xxxxxx	
4 Bytes	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

The Go Unicode Package

1. Defines a number of classes used in Unicode (eg. whitespace).
2. Also defines what Unicode codes points are used by each language.
3. Provides a number of test predicates like IsLetter() what class a code point belongs to.
4. Provides conversion functions like ToUpper() that makes language specific conversions.
5. Generally the unicode package has functions that are analogous to the functions in "strings" but they work on non-ascii Unicode points.

Example 14-04

```
1 // Example 14-04 The unicode package
2
3 package main
4
5 import (
6     "fmt"
7     "unicode"
8 )
9
10 func main() {
11
12     const french string = "語. Côté"
13     for index, runeValue := range french {
14         fmt.Printf("%c starts at byte position %d with length\n", runeValue, index)
15         fmt.Printf(" %c is uppercase? %t\n", runeValue, unicode.IsUpper(runeValue))
16         fmt.Printf(" %c is white space? %t\n", runeValue, unicode.IsSpace(runeValue))
17         fmt.Printf(" %c is letter? %t\n", runeValue, unicode.IsLetter(runeValue))
18     }
19 }
20 }
```

Example 14-04

```
/usr/local/go/bin/go run ex14-04.go [/home/gospace/Module14]
```

```
語 starts at byte position 0 with length
```

```
    語 is uppercase? false
```

```
    語 is white space? false
```

```
    語 is letter? true
```

```
. starts at byte position 3 with length
```

```
    . is uppercase? false
```

```
    . is white space? false
```

```
    . is letter? false
```

```
starts at byte position 4 with length
```

```
    is uppercase? false
```

```
    is white space? true
```

```
    is letter? false
```

```
C starts at byte position 5 with length
```

```
    C is uppercase? true
```

```
    C is white space? false
```

```
    C is letter? true
```

```
ô starts at byte position 6 with length
```

```
    ô is uppercase? false
```

```
    ô is white space? false
```

```
    ô is letter? true
```

```
t starts at byte position 8 with length
```

```
    t is uppercase? false
```

```
    t is white space? false
```

```
    t is letter? true
```

```
é starts at byte position 9 with length
```

```
    é is uppercase? false
```

```
    é is white space? false
```

```
    é is letter? true
```

```
Success: process exited with code 0.
```

The Go utf8 Package

1. Unicode code points are called runes in Go.
2. The utf8 package has low level functions that allow us to both encode runes (which are 4 bytes long) into variable length utf-8 encodings and vice versa.
3. Also provides predicates to count the number of runes in a strings, extract runes from strings and validate if a sequence of bytes is a utf-8 encoding of a rune.
4. There is also a utf16 package which supplies the same functionality for utf-16 encoded strings (the format used by Java).

Example 14-05

```
1 // Example 14-05 The unicode/utf8 package
2 // Demonstrating utility functions.
3
4 package main
5
6 import (
7     "fmt"
8     "unicode/utf8"
9 )
10
11 func main() {
12
13     const test1 string = "\xff\xff"
14     fmt.Printf("%s is valid Unicode? %t\n", test1, utf8.ValidString(test1))
15
16     const test2 string = "Côté"
17     fmt.Printf("%s is %d bytes and %d runes long\n", test2, len(test2),
18                 utf8.RuneCountInString(test2))
19
20 }
```

Build Output ▾ ■ 🔍 ⚙

/usr/local/go/bin/go run ex14-05.go [/home/gospace/Module14]

ffff is valid Unicode? false

Côté is 6 bytes and 4 runes long

Success: process exited with code 0.

Example 14-06

```
1 // Example 14-06 The unicode(utf8) package
2 // Encoding and Decoding Runes
3
4 package main
5
6 import (
7     "fmt"
8     "unicode/utf8"
9 )
10
11 func main() {
12
13     // creating a rune
14     buf := []byte{228, 184, 150}
15     r, size := utf8.DecodeRune(buf)
16     fmt.Printf("Rune r is %c and is %d bytes wide\n", r, size)
17
18     // converting rune to bytes
19     buf2 := make([]byte, 3)
20     _ = utf8.EncodeRune(buf2, r)
21     fmt.Println(buf2)
22 }
23
```

Build Output ▾

/usr/local/go/bin/go run ex14-06.go [/home/gospace/Module14]

Rune r is 世 and is 3 bytes wide

[228 184 150]

Success: process exited with code 0.

Pattern Matching and Regular Expressions

Compiling Regular Expressions

1. Go supports regular expressions as most programming languages do.
2. Pattern matching can be done with simple regex strings as patterns.
3. However, Go can compile patterns into regular expression objects.
4. Compiled regular expression objects guaranteed to run in linear time with respect to input size.
5. Compiled regular expression objects can also be used for more advanced functionality.
6. Compiling a regular expression pattern will raise an error if the string being compiled is not a valid regex expression.

Example 14-07

```
1 // Example 14-07 Regular Expressions
2 package main
3 import (
4     "fmt"
5     "regexp"
6 )
7
8 func main() {
9
10    target := "Amal Singer: (613) 555-1212"
11
12    pattern := "6[0-9]+3"
13
14    regexObject, err := regexp.Compile(pattern)
15    if err != nil {
16        fmt.Printf("ERROR |%s| is not a valid regex", pattern)
17    } else {
18        fmt.Printf("Pattern /%s/ found in \"%s\" is %t \n", pattern, target,
19                    regexObject.MatchString(target))
20        fmt.Printf("Matching string is %s \n", regexObject.FindString(target))
21    }
22
23 }
```

Build Output ▾

/usr/local/go/bin/go run ex14-07.go [/home/gospace/Module14]

Pattern /6[0-9]+3/ found in "Amal Singer: (613) 555-1212" is true

Matching string is 613

Success: process exited with code 0.

Basic Matching

1. The `MatchString()` method returns a Boolean if the pattern is found in the target string.
2. The `FindString()` method returns a string which is the first match found to the pattern found in the target.
3. The `Match()` function returns a boolean if a pattern string is found in a byte slice representing a string, or an error if the pattern is not a valid regex.

Example 14-08

```
1 // Example 14-08 Regular Expressions without Comiling
2
3 package main
4
5 import (
6     "fmt"
7     "regexp"
8 )
9
10 func main() {
11
12     str1 := "Amal Singer: (613) 555-1212"
13
14     pattern := "6[0-9]+3"
15
16 if ok, _ := regexp.Match(pattern, []byte(str1)); ok {
17     fmt.Println("Match found!")
18 }
19 }
20
```

Build Output



```
/usr/local/go/bin/go run ex14-08.go [/home/gospace/Module14]
```

```
Match found!
```

```
Success: process exited with code 0.
```

Finding All Matches

1. The FindAllStrings(target,num) returns a slice of strings that contain one or more matches of the pattern in the target.
2. If the number provided is -1, then all matches are returned.
3. If the number is positive, then that number of matches from the start of the target are returned.

Example 14-09

```
1 // Example 14-09 Finding Substrings
2
3 package main
4
5 import (
6     "fmt"
7     "regexp"
8 )
9
10 func main() {
11
12     target := "Amal 555-1212 : Josh 247-1133 : Vu 998-7734"
13
14     pattern := "[0-9]+-[0-9]+"
15
16     rObj, _ := regexp.Compile(pattern)
17     fmt.Println(rObj.FindAllString(target, -1)) // return all matches
18     fmt.Println(rObj.FindAllString(target, 2)) // only return two matches
19     fmt.Println(rObj.FindAllString("192.168.1.1 logged off", -1))
20 }
```

Build Output ▾ ■ 🔍 ⚙

```
/usr/local/go/bin/go run ex14-09.go [/home/gospace/Module14]
[555-1212 247-1133 998-7734]
[555-1212 247-1133]
[]
Success: process exited with code 0.
```

Replacing Matches

1. ReplaceAllString(target,mask) as the effect of replacing every matching sub-string in the target, no matter how long, with the mask string.
2. Because it is not a straight string replacement, the mask may replace sub-strings that are different lengths than the mask string.
3. This sort of matching is often used to anonymize data.

Example 14-10

```
1 // Example 14-10 Replacing Substrings
2
3 package main
4
5 import (
6     "fmt"
7     "regexp"
8 )
9
10 func main() {
11
12     target := "Amal 555-1212 : Josh 247-1133 : Vu 998-7734"
13
14     pattern := "[0-9]+-[0-9]+"
15
16     rObj, _ := regexp.Compile(pattern)
17     fmt.Println(rObj.ReplaceAllString(target, "XXX-XXXX"))
18 }
```

Build Output ▾ ■ 🔍 ⚙

/usr/local/go/bin/go run ex14-10.go [/home/gospace/Module14]

Amal XXX-XXXX : Josh XXX-XXXX : Vu XXX-XXXX

Success: process exited with code 0.

Data-String Conversions

Package Initialization

1. Data-string conversions are necessary when converting Go data types (int, float32, etc) to strings for transmission or using pattern matching or text file input and output.
2. The most common conversion is converting strings into integers, called parsing, done by the Atoi() method; and
3. Converting integers to strings, called formatting, done the Iota() method.

Example 14-11

```
1 // Example 14-11 Atoi and Itoa
2
3 package main
4
5 import (
6     "fmt"
7     "strconv"
8 )
9
10 func main() {
11     i, _ := strconv.Atoi("-42")
12     s := strconv.Itoa(-42)
13     fmt.Printf("i is of type %T, s is of type %T\n", i, s)
14
15     i, err := strconv.Atoi("-42 degrees")
16     if err != nil {
17         fmt.Println("Atoi method failed")
18     }
19 }
```

Build Output

```
/usr/local/go/bin/go run ex14-11.go [/home/gospace/Module14]
```

```
i is of type int, s is of type string
```

```
Atoi method failed
```

```
Success: process exited with code 0.
```

Formatting Functions

1. The Formatxxx() functions all convert a numeric or boolean into a string.
2. The FormatInt() needs to have the base of the integer provided.
3. Since FormatInt() only takes int64 types as arguments so any value needs to be converted to an int64 before use.
4. FormatUint() works the same way.
5. FormatFloat() needs to have the precision, exponent format and size specified.

Example 14-12

```
1 // Example 14-12 Formatting Numerics
2
3 package main
4
5 import (
6     "fmt"
7     "strconv"
8 )
9
10 func main() {
11
12     var i int32 = 255
13     var f float64 = 3.14159
14     fmt.Println(strconv.FormatInt(int64(i), 10))
15     fmt.Println(strconv.FormatInt(int64(i), 16))
16     // -1 means use the smallest precision possible
17     fmt.Println(strconv.FormatFloat(f, 'E', -1, 32))
18     fmt.Println(strconv.FormatFloat(f, 'f', 10, 64))
19     fmt.Println(strconv.FormatBool(true))
20 }
```

Build Output ▾ ■ 🔍 ⚙

```
/usr/local/go/bin/go run ex14-12.go [/home/gospace/Module14]
255
ff
3.14159E+00
3.1415900000
true
Success: process exited with code 0.
```

Parsing Functions

1. Parsing functions convert a string representation into a Go data type.
2. Parsing functions can throw an error since a string may not represent a valid numeric.
3. For integers we generally have to provide the base of the integer represented in the string, if the value 0 is provided, then the prefix (eg 0x) is used if present otherwise base 10 is assumed.
4. We also have to specify the size of the integer data type the value should be returned

Example 14-13

```
1 // Example 14-13 Parsing Numerics
2 ...
3 func main() {
4
5     var s1 string = "7386"
6     var s2 string = "One Hundred"
7     var s3 string = "0xff"
8     var s4 string = "7889272162938"
9     if i1, err := strconv.ParseInt(s1, 10, 16); err == nil {
10         fmt.Printf("%T, %v\n", i1, i1)
11     }
12     if i1, err := strconv.ParseInt(s2, 10, 16); err == nil {
13         fmt.Printf("%T, %v\n", i1, i1)
14     } else {
15         fmt.Println("Parse Error")
16     }
17     if i1, err := strconv.ParseInt(s3, 0, 16); err == nil {
18         fmt.Printf("%T, %v\n", i1, i1)
19     }
20     if i1, err := strconv.ParseInt(s4, 10, 16); err == nil {
21         fmt.Printf("%T, %v\n", i1, i1)
22     } else {
23         fmt.Println("Parse Error")
24     }
25 }
```

Build Output ▾

```
/usr/local/go/bin/go run ex14-13.go [/home/gospace/Module14]
int64, 7386
Parse Error
int64, 255
Parse Error
Success: process exited with code 0.
```

File and Terminal I/O

Stdin and Stdout

1. Go uses the standard files stdin, stdout and stderr accessed through the os package as os.Stdin, etc
2. Input is written to the console using the fmt.Println() functions using os.Stdout in the background.
3. Input is fetched from the console using Scanxx() functions.
4. The scan functions work the analogous to the C style scan functions.
5. Scanln() tokenizes the input from os.Stdin until it encounters a \n.
6. Scan() does the same as Scanln() but treats \n as whitespace.
7. Scanf() does the same as a Scanln() but the first argument is taken to be a Printf() style formatting string.

Example 14-14

```
1 // Example 14-14 Scanln() and Scanf()
2
3 package main
4
5 import (
6     "fmt"
7 )
8
9 func main() {
10
11     var firstName, lastName string
12     var age int
13
14     fmt.Println("Please enter your full name: ")
15     fmt.Scanln(&firstName, &lastName)
16     fmt.Println("Please enter your age: ")
17     fmt.Scanf("%d", &age)
18     fmt.Printf("Hi %s %s! you are %d years old \n", firstName, lastName, age)
19 }
20
```

Build Output ▾ ■ 🔍 ⚙

/usr/local/go/bin/go run ex14-14.go [/home/gospace/Module14]

Please enter your full name:

Albert Einstein

Please enter your age:

187

Hi Albert Einstein! you are 187 years old

Success: process exited with code 0.

Reading from Strings

1. The Sscanf() functions work exactly like the Scanf() functions except that they read from strings

Example 14-15

```
1 // Example 14-15 Sscanfln() and Sscanf()
2
3 package main
4
5 import (
6     "fmt"
7 )
8
9 func main() {
10
11     var firstName, lastName string
12     var age int
13     var nameInput string = "Albert Einstein"
14     var ageInput string = "100"
15
16     fmt.Sscanfln(nameInput, &firstName, &lastName)
17     fmt.Sscanf(ageInput, "%d", &age)
18     fmt.Printf("Hi %s %s! you are %d years old \n", firstName, lastName, age)
19 }
```

Build Output

/usr/local/go/bin/go run ex14-15.go [/home/gospace/Module14]

Hi Albert Einstein! you are 100 years old

Success: process exited with code 0.

Opening and Closing Files

1. Files in Go are pointers to objects of type `os.File` which are structs that contain information to access the underlying OS file handles
2. The `os.Open()` method creates the file struct and returns an error if the file cannot be opened for any reason.
3. Open files should be closed when we are done with them -- the Go style is to defer the `os.Close()` call to ensure the file is always closed.

Example 14-16

```
1 // Example 14-16 Opening and Closing Files
2
3 package main
4
5 import (
6     "fmt"
7     "os"
8 )
9
10 func main() {
11
12     filename := "input.txt"
13
14     myFile, err := os.Open(filename)
15     if err != nil {
16         fmt.Printf("Cannot open the file %s for input\n", filename)
17         return // exit the function on error
18     }
19     fmt.Println(myFile)
20     defer myFile.Close()
21
22 }
```

Build Output ▾

```
/usr/local/go/bin/go run ex14-16.go [/home/gospace/Module14]
&{0xc42000a520}
Success: process exited with code 0.
```

Buffered Reading

1. Wrapping the file object in a Reader object allows us to treat the file I/O as if we were working with strings.
2. ReadLine() read a file line by line where lines are terminated with \n.
3. ReadString(delim) reads a line up to a the delimiter delim.
4. Read() uses a buffer and reads bytes into the buffer.
5. Read() returns the number of bytes read or 0 if an EOF is encountered.

Example 14-17

```
1 // Example 14-17 Buffered ReadLine()
2
3 package main
4
5 import (
6     "bufio"
7     "fmt"
8     "io"
9     "os"
10)
11
12 func main() {
13
14     filename := "input.txt"
15
16     myFile, err := os.Open(filename)
17     if err != nil {
18         return
19     }
20     defer myFile.Close()
21
22     reader := bufio.NewReader(myFile)
23
24     for {
25         line, err := reader.ReadString('\n')
26         if err == io.EOF {
27             return
28         }
29         fmt.Printf("File line: %s", line)
30     }
31 }
```

Build Output

```
/usr/local/go/bin/go run ex14-17.go [/home/gospace/Module14]
File line: This is the first line in the file
File line: This is the second line in the file
Success: process exited with code 0.
```

Example 14-18

```
12 func main() {
13
14     filename := "input.txt"
15
16     myFile, err := os.Open(filename)
17     if err != nil {
18         return
19     }
20     defer myFile.Close()
21
22     reader := bufio.NewReader(myFile)
23     buffer := make([]byte, 1024)
24
25     for {
26         n, err := reader.Read(buffer)
27         if err == io.EOF {
28             break
29         }
30         fmt.Printf("File (%d bytes) is:\n%s", n, string(buffer))
31     }
32 }
```

Build Output ▾

/usr/local/go/bin/go run ex14-18.go [/home/gospace/Module14]

File (71 bytes) is:

This is the first line in the file

This is the second line in the file

Success: process exited with code 0.

Reading a Whole File

1. The iotools package allows an entire file to be read into a buffer.
2. The function does not need a file object.
3. The function will return an error if the file cannot be opened.

Example 14-19

```
1| // Example 14-19 ReadFile()
2
3 package main
4
5 import (
6     "fmt"
7     "io/ioutil"
8 )
9
10 func main() {
11
12     filename := "input.txt"
13
14     buffer, err := ioutil.ReadFile(filename)
15     if err != nil {
16         fmt.Println("Unable to read from file")
17         return
18     }
19     fmt.Printf("File is: %s", string(buffer))
20 }
```

Build Output

```
usr/local/go/bin/go run ex14-19.go [/home/gospace/Module14]
File is: This is the first line in the file
This is the second line in the file
Success: process exited with code 0.
```