UNIT 9:

Standard Library

- fmt
- net/http
- encoding/json
- regexp
- strconv
- errors
- log

Builtins

- The builtin **constants**, **variables**, **types**, and **functions** provided by Go are listed as being in the standard library package "builtin" for documentation purposes
- But no such package actually exists
- Following sections describe each of the provided builtins

Builtin Constants

- The provided constants include the boolean literals `true` and `false`, and `iota`
- `iota` is not actually a constant
  - It is a global counter that is set to zero at the beginning of every const definition, which is the only place it can be used
- The value of `iota` is incremented by one after each line in the `const` definition, except for blank lines and comment lines
  - It is typically used to define enumerated values

Builtin Constants

- The last expression involving `iota` is repeated for subsequent constant values but uses an incremented value of `iota`
- For example:

```
1  const (
2    red   = iota // 0
3    green        // 1
4    blue         // 2
5  )
6
7  const (
8    north = iota + 1 // iota = 0, 0 + 1 = 1
9    south            // iota = 1, 1 + 1 = 2
10   east             // iota = 2, 2 + 1 = 3
11   west             // iota = 3, 3 + 1 = 4
12 )
13
14 const (
15   t1 = iota * 3 // iota = 0, 0 * 3 = 0
16   t2            // iota = 1, 1 * 3 = 3
17   t3            // iota = 2, 2 * 3 = 6
18 )
19
```

Builtin Constants

```
20  const (
21          _  = iota             // iota = 0, ignore first value
22    kb int64 = 1 << (10 * iota) // iota = 1, 1 shifted left 10 places, 1024
23    mb                          // iota = 2, 1 shifted left 20 places,
          1048576
24    gb                          // iota = 3, 1 shifted left 30 places,
          1073741824
25  )
26
27  // Silly example
28  const (
29    apple  = 9         // iota = 0
30    banana = 8         // iota = 1
31    cherry = iota + 3  // iota = 2, value = 2 + 3 = 5
32    date               // iota = 3, value = 3 + 6 = 6
33  )
```

Builtin Variables

- There is only one provided variable, named `nil`
- This is the zero value for a pointer, channel, func, interface, map, or slice
- Example, the current value of all the variables below is nil

```
1  var ptr *string          // pointer
2
3  var c chan string        // channel
4
5  type stringToString func(string) string
6  var f stringToString     // function
7
8  var i error              // interface
9
10 var m map[string]string // map
11
12 var s []string           // slice
```

Builtin Types

- Go defines the following builtin "basic types."
- `bool` : The only values are the builtin constants `true` and `false`
  - These can be used with the operators `&&`, `||`, and `!`
- `byte` : This is an alias for the type `uint8`
- `complex64` and `complex128` : These are used to represent complex numbers with a specified number of bits
- `float32` and `float64` : These are used to represent floating-point numbers with a specified number of bits
  - `float64` is preferred in most cases

Builtin Types

- `int, int8, int16, int32, int64 :` These are used to represent signed integers with a specified number of bits
  - The type `int` is at least 32 bits
  - Its size is based on the word size of the host platform, 32 bits on 32-bit systems and 64 bits on 64-bit systems
  - `int` is preferred in many cases
- `uint, uint16, uint32, uint64` : These are used to represent unsigned integers with a specified number of bits
  - The type `uint` is at least 32 bits

Builtin Types

- `uintptr` : This type can hold any kind of pointer
- `rune` : This is an alias for `int32`
  - It is used for unicode characters that range in size from 1 to 4 bytes (a.k.a. Unicode code point)
  - Literal values of this type are surrounded by single quotes
- `string` : This is a sequence of 8-bit bytes, not Unicode characters
  - However, the bytes are often used to represent Unicode characters

Builtin Types

- Go defines the type error to represent an `error` condition
  - Variables of this type have the value `nil` when there is no error
- Non-basic types include aggregate, reference, and interface types
- Aggregate types include arrays and structs
- Reference types include pointers, slices, maps, functions, and channels

Documentation Types

- Despite the fact that Go does not currently support generic types, the following "generic type" names appear in the Go documentation
  - `Type` – represents a specific type for a given function invocation
  - `Type1` – like `Type` but for a second type
  - `ComplexType` – represents a `complex64` or `complex128`
  - `FloatType` – represents a `float32` or `float64`
  - `IntegerType` – represents any integer type

Builtin Functions

- **Data Structure Functions**
- `append(slice []Type, elems ...Type) []Type` : This appends elements to the end of a slice and returns the updated slice
- `cap(v Type) int` : This returns the capacity of a string, array, slice, or map
- `copy(dst, src []Type) int` : This copies elements from a source slice to a destination slice and returns the number of elements copied
- `delete(m map[Type]Type1, key Type)` : This deletes a key/value pair from a map

Builtin Functions

- `len(v Type) int` : This returns the length of a string, array, slice, or map
- `make(t Type, size ...IntegerType) Type:` This allocates and initializes a slice, map, or channel
  - If Type is Slice, pass the length, and optional capacity
  - If Type is Map, optionally specify the number of key/value pairs for which to allocate space
- `new(Type) *Type` : This allocates memory for a given type and returns a pointer to it

Builtin Functions

- **Output Functions**
- `print(args ...Type)` : This writes to stderr; useful for debugging
- `println(args ...Type)` : This is like `print` but adds a newline at the end
- **Error Handling Functions**
- `panic(v interface{})` : This stops normal execution of the current goroutine
  - It is somewhat like a `throw` in other languages

Builtin Functions

- Control cascades upward through the call stack
- When it reaches the top, the program is terminated and an error is reported
- This can be controlled by the `recover` function

➢ `recover` : This should be called inside a deferred function to stop the panic sequence and restore normal execution

- It is somewhat like a catch in other languages

Builtin Functions

- **Channel Functions**
- `close(c chan<-)` : This closes a channel after the last sent value is received
- `make(Channel [, buffer-capacity])` : This creates a channel
  - The channel is unbuffered if `buffer-capacity` is omitted or is zero

Builtin Functions

- **Complex Number Functions**
- `complex(real, imag FloatType) ComplexType` : This creates a complex value from two floating-point values that represent the real and imaginary parts
- `imag(c ComplexType) FloatType` : This returns the imaginary part of a complex number
- `real(c ComplexType) FloatType` : This returns the real part of a complex number

Standard Library Packages

- Go provides many packages in the "standard library."
- `bufio` : This provides functions to perform buffered I/O using the types `Reader` and `Writer`
  - It also provides a `Scanner` type that splits input into lines and words
- `builtin` : This not a real package, just a place to document builtin constants, variables, types, and functions

Standard Library Packages

- `container/heap` : This implements a kind of tree data structure
- `container/list` : This implements a doubly linked list
- `container/ring` : This implements circular lists
- `database/sql` : This defines interfaces implemented by relational database-specific drivers
  - For example, there are drivers for MySQL and PostgreSQL
- `encoding` : This defines interfaces for reading and writing various data formats such as CSV, JSON, and XML

Standard Library Packages

- `errors` : This provides the New function that creates error values that have a string description and a method named Error to retrieve the description
- `flag` : This provides flag parsing for command-line applications
- `fmt` : This provides functions for formatted I/O
  - Many of its functions are similar to C's printf and scanf
- `go` : The sub-packages of this package implement all the standard Go tooling, such as source file parsing to ASTs and code formatting

Standard Library Packages

- `html` : This provides functions to parse and create HTML
- `image` : This provides functions to parse (decode) and create (encode) images in GIF, JPEG, and PNG formats
- `io` : This provides functions to read and write buffers and files
  - The function io.Copy copies data from a writer to a reader
- `log` : This provides simple logging
- `math` : This provides many math functions, including ones for logarithms and trigonometry

Standard Library Packages

- `mime` : This provides functions to encode and decode multimedia formats
- `net` : This provides functions that perform network I/O, including TCP and UDP
- `net/http` : This provides functions to send and listen for HTTP and HTTPS requests
- `os` : This provides access to operating system functionality like that provided by UNIX shell commands

Standard Library Packages

- It defines the File type, which supports opening, reading, writing, and closing files
- It defines the constants PathSeparator ('/' on UNIX), and PathListSeparator (':' on UNIX)
- It provides the function os.Exit(status) that exits the process with a given status

Standard Library Packages

- `os/exec` : This provides functions that run external commands
- `path` : This provides functions that work with UNIX-style file paths and URLs
- `reflect` : This provides types and functions that support using reflection to work with types determined at runtime
- `regexp` : This provides functions that perform regular expression searches
- `sort` : This provides functions that sort slices and other collections

Standard Library Packages

- `strconv` : This provides conversions to and from string representations of primitive types
  - For example, `strconv.Atoi` converts a `string` to an `int`, and `strconv.Itoa` converts an `int` to a `string`
- `strings` : This provides many functions that operate on strings, including `Contains`, `HasPrefix`, `HasSuffix`, `Index`, `Join`, `Repeat`, `Split`, `ToLower`, `ToTitle`, `ToUpper`, and `Trim`
  - It also defines the `Builder`, `Reader`, and `Replacer` types

Standard Library Packages

- `sync` : This provides synchronization primitives, such as mutual exclusion locks
  - Often code will use channels and select instead to achieve this
- `testing` : This provides functions and types that support automated tests run by go test
  - The sub-package quick implements fuzz testing

Standard Library Packages

- `text` : This provides functions that parse text, write tabbed columns, and support data-driven templates
- `time` : This provides functions that measure and display times and dates
- `unicode` : This provides functions that work with and test Unicode characters

Formatting

- The standard library package `fmt` defines many functions that read and write formatted messages
- Functions that read have names that start with `Scan`
- Functions that write have names that start with `Print`
- The most commonly used functions in this package include:
- `fmt.Errorf(format string, args ...interface{}) error`: This creates an error value containing a formatted message.
- `fmt.Printf(format string, args ...interface{}) :` This writes a formatted string to stdout

Formatting

- `fmt.Println(args ...interface{})` : This writes the string representation of each of the arguments to stdout, separated by spaces and followed by a newline
- Format strings can contain placeholders that begin with a percent sign
- Commonly used verbs include:
  - `%d` for decimal values (includes all the integer types)
  - `%f` for floating point values
  - `%s` for strings

Formatting

- `%t` for boolean values to output "true" or "false"
- `%p` for pointers (prints hex address of a variable)
- `%v` for any value in its default format
- `%+v` is similar to %v but includes struct field names
- `%T` to output the type of a value

Example

Execute | Share main.go STDIN

```go
package main

import (
    "fmt"

)

func main() {
  fmt.Printf("[%*s]\n", 5, "abc") // [  abc], right-aligned by default
fmt.Printf("[%-*s]\n", 5, "abc") // [abc  ], left-aligned by dash
fmt.Printf("[%*s]\n", 3, "abcdef") // [abcdef], not truncated

fmt.Printf("[%*d]\n", 5, 123) // [  123]
fmt.Printf("[%*d]\n", 3, 12345) // [12345], not truncated

fmt.Printf("[%*.2f]\n", 5, 3.456) // [ 3.46]
fmt.Printf("[%*.*f]\n", 5, 2, 3.456) // outputs same
}
```

Result

```
$go run main.go
[  abc]
[abc  ]
[abcdef]
[  123]
[12345]
[ 3.46]
[ 3.46]
```

JSON

- The encoding/json standard library package supports marshaling and unmarshaling of JSON data
- Go arrays and slices are represented by JSON arrays
- Go structs and maps are represented by JSON objects
- The encoding/xml standard library package provides similar functionality for XML
- To marshal data to JSON, use the json.Marshal function
- This takes a Go value and returns a byte slice that can be converted to a string with the string function

Logging

- The standard library package `log` provides functions that help with writing error messages to stderr
- By default, `log.Fatal(message)` outputs a line containing the date, time, and message, and exits with a status code of 1
- By default, `log.Fatalf(formatString, args)` is similar, but uses a format string to specify a message string that includes placeholders for the remaining arguments
- The date and time can be suppressed in all messages produced by the log package by calling log.SetFlags(0)

Logging

- This function takes an integer, which is the result of or'ing predefined constants that identify the desired parts of the prefix
- The constants are:
  - `Ldate` – yyyy/mm/dd in local time zone
  - `Ltime` – hh:mm:ss in local time zone
  - `Lmicroseconds` – hh:mm:ss.microseconds
  - `Llongfile` – full-file-path:line-number
  - `Lshortfile` – file-name.file-extension:line-number
  - `LUTC` – use UTC instead of local time zone for dates and times

Regular Expressions

- The standard library package regexp defines functions and the type Regexp for working with regular expressions
- The easiest way to determine if text matches a regular expression is to use the functions `MatchString` and `Match`
- Both return a `bool` indicating whether there is a match
- These differ in how the text to be tested is supplied
- `MatchString` takes a `String`, and `Match` takes a byte slice

Regular Expressions

Execute | Share main.go STDIN

```go
package main

import (
  "fmt"
  "io/ioutil"
  "log"
  "regexp"
)

func main() {
  text := "FooBarBaz"
  matched, err := regexp.MatchString("Bar", text)
  fmt.Println(matched, err) // true nil
  matched, err = regexp.MatchString("^Foo", text)
  fmt.Println(matched, err) // true nil
  matched, err = regexp.MatchString("Baz$", text)
  fmt.Println(matched, err) // true nil
  matched, err = regexp.MatchString("bad[", text)
  fmt.Println(matched, err) // false error parsing regexp: missing closing ]

  // The file haiku.txt contains:
  // Out of memory.
  // We wish to hold the whole sky,
  // but we never will.
  bytes, err := ioutil.ReadFile("haiku.txt")
  if err != nil {
    log.Fatal(err)
  }
  matched, err = regexp.Match("whole sky", bytes)
  fmt.Println(matched, err) // true nil
}
```

Result

```
$go run main.go
true <nil>
true <nil>
true <nil>
false error parsing regexp: missing closing ]: `[`
2022/01/30 15:08:55 open haiku.txt: no such file or directory
exit status 1
```