Unit : 6

Packaging

- Naming conventions
- Importing
- Visibility rules
- Documentation
- Init functions and package lifecycle

Naming conventions

- A name must begin with a letter, and can have any number of additional letters and numbers.
- A function name cannot start with a number.
- A function name cannot contain spaces.
- If the functions with names that start with an uppercase letter will be exported to other packages.

Naming conventions

- If the function name starts with a lowercase letter, it won't be exported to other packages, but you can call this function within the same package
- If a name consists of multiple words, each word after the first should be capitalized like this: empName, EmpAddress, etc
- function names are case-sensitive (car, Car and CAR are three different variables)

Importing

- **Direct import**
- Go supports the direct import of packages by following a simple syntax
- Both single and multiple packages can be imported one by one using the import keyword

Importing

➢ **Example:**

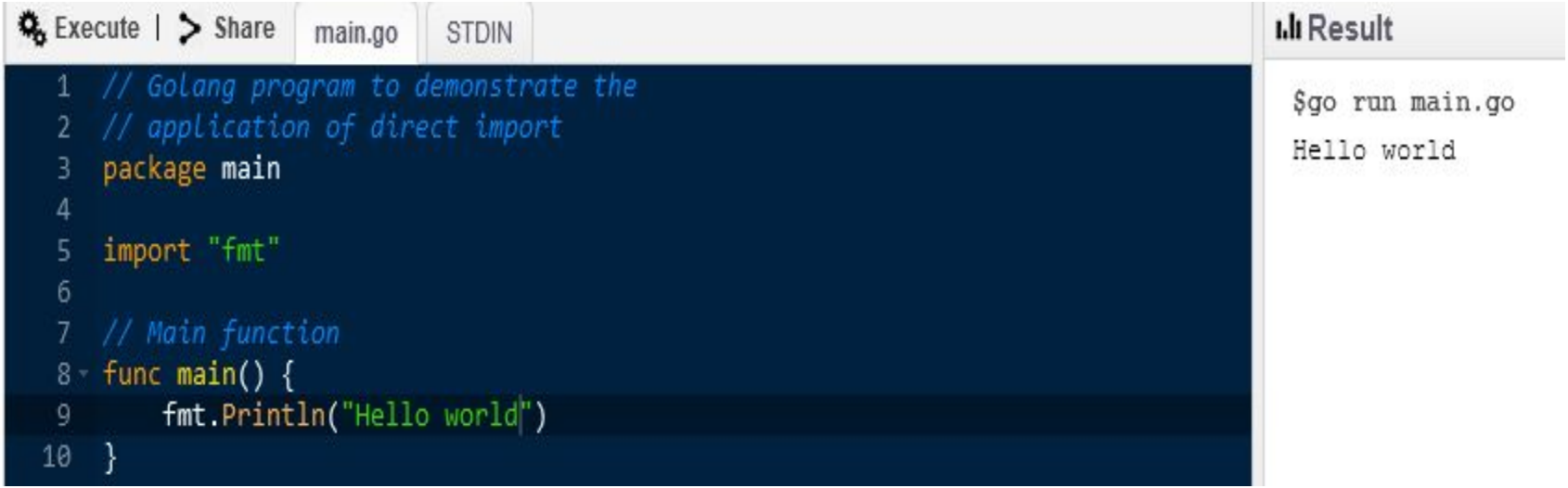
Single:

```
import "fmt"
```

➢ **Multiple one-by-one:**

```
import "fmt"
    import "math"
```

Example


Execute | Share
main.go
STDIN
1 // Golang program to demonstrate the
2 // application of direct import
3 package main
4
5 import "fmt"
6
7 // Main function
8 func main() {
9 fmt.Println("Hello world")
10 }
Result
$go run main.go
Hello world

Grouped import

- Go also supports grouped imports.
- This means that you don’t have to write the import keyword multiple times; instead, you can use the keyword import followed by round braces, ()
- And mention all the packages inside the round braces that you wish to import

Grouped import

- This is a direct import too but the difference is that you've mentioned multiple packages within one import() here
- Peek at the example to get an idea of the syntax of grouped import command
- **Example:**

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

Example

Execute | Share main.go STDIN

```go
// A program to demonstrate the
// application of grouped import
package main

import (
    "fmt"
    "math"
)

// Main function
func main() {

    // math.Exp2(5) returns
    // the value of 2^5, wiz 32
    c := math.Exp2(5)

    // Println is a function in fmt package
    // which prints value of c in a new
    // line on console
    fmt.Println(c)

}
```

Result

```
$go run main.go
32
```

Nested import

- Go supports nested imports as well
- Just as we hear of the name nested import, we suddenly think of the ladder if-else statements of nested loops, etc
- But nested import is nothing of that sort: It’s different from those nested elements
- Here nested import means, importing a sub-package from a larger package file

Nested import

- For instance, there are times when you only need to use one particular function from the entire package and so you do not want to import the entire package and increase your memory size of code and stuff like that, in short, you just want one sub-package
- In such scenarios, we use nested import
- Look at the example to follow syntax and example code for a better understanding
- Example : import "math/rand"

Example

Execute | Share main.go STDIN

```go
// Golang Program to demonstrate
// application of nested import
package main

 import (
    "fmt"
    "math/rand"
)

func main() {

    // this generates & displays a
    // random integer value < 100
    fmt.Println(rand.Int(100))
}
```

Result

```
$go run main.go
# command-line-arguments
./main.go:14: too many arguments in call to rand.Int
        have (number)
        want ()
```

Aliased import

- It supports aliased imports as well
- Well at times we're just tired of writing the full name again and again in our code as it may be lengthy or boring or whatsoever and so you wish to rename it
- Alias import is just that
- It doesn't rename the package but it uses the name that you mention for the package and creates an alias of that package, giving you the impression that the package name has been renamed

Aliased import

- Consider the example for syntax and example code for a better understanding of the aliased import
- **Example:**

```
import m "math"

import f "fmt"
```

Example

Execute | Share main.go STDIN

```go
// Golang Program to demonstrate
// the application of aliased import
package main

import (
    f "fmt"
    m "math"
)

// Main function
func main() {

    // this assigns value
    // of 2^5 = 32 to var c
    c := m.Exp2(5)

    // this prints the
    // value stored in var c
    f.Println(c)
}
```

Result

```
$go run main.go
32
```

Dot import

- Go supports dot imports
- Dot import is something that most users haven’t heard of
- It is basically a rare type of import that is mostly used for testing purposes
- Testers use this kind of import in order to test whether their public structures/functions/package elements are functioning properly
- Dot import provides the perks of using elements of a package without mentioning the name of the package and can be used directly

Dot import

- As many perks as it provides, it also brings along with it a couple of drawbacks such as namespace collisions
- **Example:**

```
import . "math"
```

Example

Execute | Share main.go STDIN

```go
// Golang Program to demonstrate
// the application of dot import
package main

import (
    "fmt"
    . "math"
)

func main() {

    // this prints the value of
    // 2^4 = 16 on the console
    fmt.Println(Exp2(4))
}
```

Result

```
$go run main.go
16
```

Blank import

- Go supports blank imports
- Blank means empty
- Many times,we do not plan of all the packages we require or the blueprint of a code that we're about to write in the future
- As a result, we often import many packages that we never use in the program
- Then Go arises errors as whatever we import in Go, we ought to use it

Blank import

- Coding is an unpredictable process, one moment we need something and in the next instance, we don’t
- But go doesn’t cope up with this inconsistency and that’s why has provided the facility of blank imports
- We can import the package and not using by placing a blank
- This way your program runs successfully and you can remove the blank whenever you wish to use that package

Example

```
➢  import _ "math"
```

Example

Execute | Share | main.go | STDIN

```go
// PROGRAM1
package main
// Program to demonstrate importance of blank import

import (
    "fmt"
    "math/rand"
)

func main() {
    fmt.Println("Hello Geeks")
}

// ------------------------------------------------------------------------------
// Program1 looks accurate and everything
// seems right but the compiler will throw an
// error upon building this code. Why? Because
// we imported the math/rand package but
// We didn't use it anywhere in the program.
// That's why. The following code is a solution.
//-------------------------------------------------------------------------------

// PROGRAM2
package main
// Program to demonstrate application of blank import

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

func main() {
    fmt.Println("Hello Geeks")
    // This program compiles successfully and
    // simply prints Hello Geeks on the console.
}
```

Result

```
$go run main.go
# command-line-arguments
./main.go:24: syntax error: non-declaration statement outside function body
./main.go:32: main redeclared in this block
        previous declaration at ./main.go:10
```

Relative import

- When we create our packages and place them in a local directory or on the cloud directory, ultimately the $GOPATH directory or within that, we find that we cannot directly import the package with its name unless you make that your $GOPATH
- So then you mention a path where the custom package is available
- These types of imports are called relative imports
- We often use this type but may not know its name (Relative import)

Example

➢ import "github.com/guides/abc"

Execute | Share | main.go | STDIN

```go
package main

import "github.com/guides/abc"

// Main function
func main() {
    // The hello function is in
    // the mentioned directory
    greet.Hello()
    // This function simply prints
    // hello world on the console screen
}
```

Result

```
$go run main.go
main.go:3:8: cannot find package "github.com/guides/abc" in any of:
        /usr/lib/golang/src/github.com/guides/abc (from $GOROOT)
        /home/cg/root/8596650/go/src/github.com/guides/abc (from $GOPATH)
```

Circular import

- Just like a circle, a loop, there exists a circular import too
- This means defining a package which imports a second package implicitly and defining the second package such that it imports the first package implicitly
- This creates a hidden loop that is called the “**import loop**“
- This type, too, is unknown by many and that is because Go does not support circular imports explicitly
- Upon building such packages, the Go compiler throws an error raising a warning: “import cycle not allowed”

Visibility rules

- Go language does not have access control modifiers such as public, protected, private, etc. like other languages.
- It controls visibility through letter case.
- If the names of defined constants, variables, types, interfaces, structures, functions, etc. are uppercase

Visibility rules

- The beginning of a letter indicates that it can be accessed or called by other packages (equivalent to public), and the non-capital beginning can only be used in the package (equivalent to private, variables or constants can also start with an underscore)

Example:

visibility/test.go

Documentation

- The Go project takes documentation seriously. Documentation is a huge part of making software accessible and maintainable
- Of course it must be well-written and accurate, but it also must be easy to write and to maintain
- Ideally, it should be coupled to the code itself so the documentation evolves along with the code
- The easier it is for programmers to produce good documentation, the better for everyone

Documentation

- Godoc parses Go source code - including comments - and produces documentation as HTML or plain text
- The end result is documentation tightly coupled with the code it documents
- For example, through godoc’s web interface you can navigate from a function’s documentation to its implementation with one click

Documentation

- Godoc is conceptually related to Python's Docstring and Java's Javadoc but its design is simpler
- The comments read by godoc are not language constructs (as with Docstring) nor must they have their own machine-readable syntax (as with Javadoc)
- Godoc comments are just good comments, the sort you would want to read even if godoc didn't exist

Documentation

- The convention is simple: to document a type, variable, constant, function, or even a package, write a regular comment directly preceding its declaration, with no intervening blank line
- Godoc will then present that comment as text alongside the item it documents
- For example, this is the documentation for the fmt package's Fprint function:

Documentation

```
// Fprint formats using the default formats for its operands and writes to w.
// Spaces are added between operands when neither is a string.
// It returns the number of bytes written and any write error encountered.
func Fprint(w io.Writer, a ...interface{}) (n int, err error) {
```

- Notice this comment is a complete sentence that begins with the name of the element it describes

Documentation

- This important convention allows us to generate documentation in a variety of formats, from plain text to HTML to UNIX man pages, and makes it read better when tools truncate it for brevity, such as when they extract the first line or sentenc
- Comments on package declarations should provide general package documentation

Documentation

- These comments can be short, like the sort package's brief description:

```
// Package sort provides primitives for sorting slices and user-defined
// collections.
package sort
```

- They can also be detailed like the gob package's overview
- That package uses another convention for packages that need large amounts of introductory documentation

Documentation

- The package comment is placed in its own file, doc.go, which contains only those comments and a package clause
- When writing package comments of any size, keep in mind that their first sentence will appear in godoc’s package list
- Comments that are not adjacent to a top-level declaration are omitted from godoc’s output, with one notable exception

Documentation

- Top-level comments that begin with the word "BUG(who)” are recognized as known bugs, and included in the “Bugs” section of the package documentation
- The “who” part should be the user name of someone who could provide more information

Init functions

- In Go, the init() function can be incredibly powerful and compared to some other languages, is a lot easier to use within your Go programs.
- These init() functions can be used within a package block and regardless of how many times that package is imported, the init() function will only be called once
- Now, the fact that it is only called once is something you should pay close attention to

Init functions

- This effectively allows us to set up database connections, or register with various service registries, or perform any number of other tasks that you typically only want to do once

Init functions

```
package main

func init() {
  fmt.Println("This will get called on main initialization")
}

func main() {
  fmt.Println("My Wonderful Go Program")
}
```

Init functions

- Notice in this above example, we've not explicitly called the init() function anywhere within our program.
- Go handles the execution for us implicitly and thus the above program should provide output that looks like this:

Init functions

```
$ go run test.go
This will get called on main initialization
My Wonderful Go Program
```

- Awesome, so with this working, we can start to do cool things such as variable initialization.

```go
package main

import "fmt"

var name string

func init() {
    fmt.Println("This will get called on main initialization")
    name = "Elliot"
}

func main() {
    fmt.Println("My Wonderful Go Program")
    fmt.Printf("Name: %s\n", name)
}
```

Init functions

- In this example, we can start to see why using the init() function would be preferential when compared to having to explicitly call your own setup functions.
- When we run the above program, you should see that our name variable has been properly set and
- whilst it's not the most useful variable on the planet, we can certainly still use it throughout our Go program.

Init functions

```
$ go run test.go
This will get called on main initialization
My Wonderful Go Program
Name: Elliot
```

package lifecycle

- Package lifecycle defines an event for an app's lifecycle.
- The app lifecycle consists of moving back and forth between an ordered sequence of stages.
- For example, being at a stage greater than or equal to StageVisible means that the app is visible on the screen.
- A lifecycle event is a change from one stage to another, which crosses every intermediate stage.

package lifecycle

- For example, changing from StageAlive to StageFocused implicitly crosses StageVisible.
- Crosses can be in a positive or negative direction.
- A positive crossing of StageFocused means that the app has gained the focus.
- A negative crossing means it has lost the focus.