

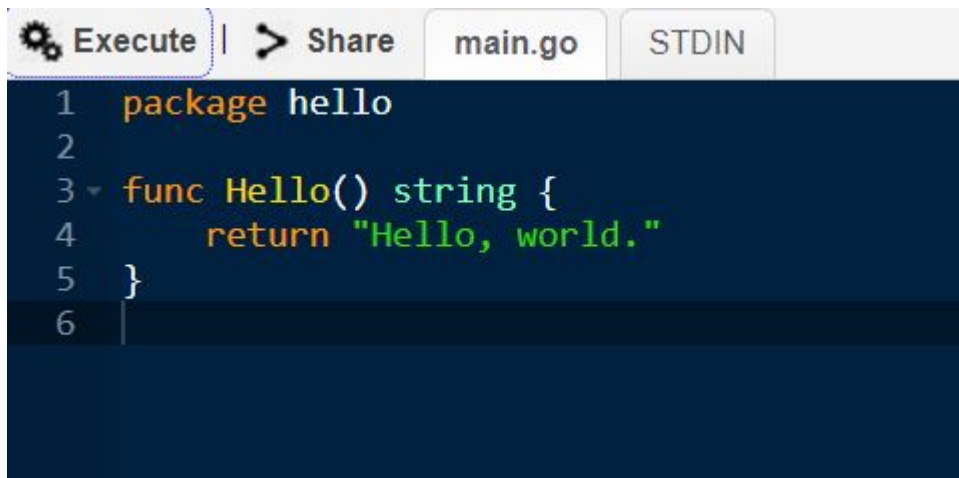
UNIT 7

MODULES

- Modules
- Creating modules
- Using external modules
- Object-Oriented Constructs
- Methods and method receivers
- Interfaces
- Type assertions
- Constructor functions

Creating modules

Create a new, empty directory somewhere outside `$GOPATH/src`, cd into that directory, and then create a new source file, `hello.go`:



The screenshot shows a code editor interface with a dark blue background. At the top, there is a toolbar with three buttons: 'Execute' (with a gear icon), 'Share' (with a right-pointing arrow icon), and 'main.go'. To the right of these buttons is a tab labeled 'STDIN'. Below the toolbar, the code is displayed in a monospaced font with syntax highlighting. The code consists of six lines: line 1 is 'package hello', line 2 is an empty line, line 3 is 'func Hello() string {' with a small dropdown arrow to the left of 'func', line 4 is ' return "Hello, world."', line 5 is '}', and line 6 is an empty line with a cursor at the end.

```
1 package hello
2
3 func Hello() string {
4     return "Hello, world."
5 }
6
```

Creating modules

Let's write a test, too, in `hello_test.go`

`package hello`



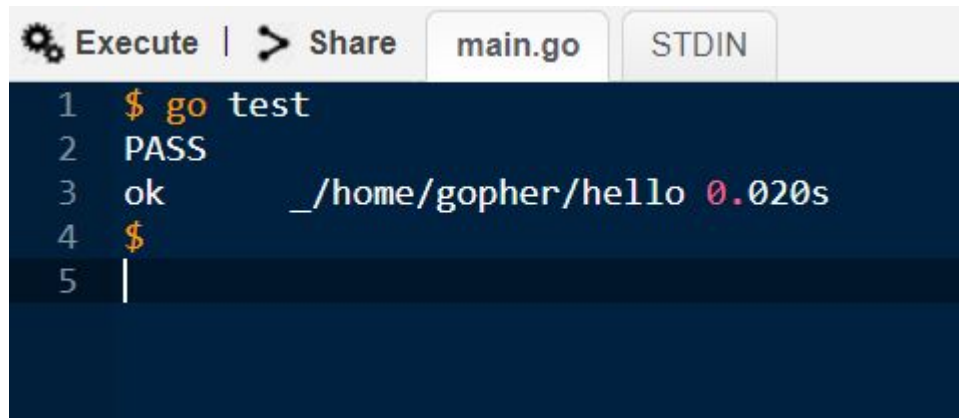
The screenshot shows a Go Playground interface. At the top, there are tabs for 'main.go' and 'STDIN'. Below the tabs, the code editor displays the following Go code:

```
1 import "testing"
2
3 func TestHello(t *testing.T) {
4     want := "Hello, world."
5     if got := Hello(); got != want {
6         t.Errorf("Hello() = %q, want %q", got, want)
7     }
8 }
```

Creating modules

At this point, the directory contains a package, but not a module, because there is no `go.mod` file.

If we were working in `/home/gopher/hello` and ran `go test` now, we'd see:



The screenshot shows a Go Playground interface with a dark blue background. At the top, there are tabs for 'Execute' (with a gear icon), 'Share' (with a right arrow icon), 'main.go', and 'STDIN'. Below the tabs, a terminal window displays the output of a `go test` command. The output is as follows:

```
1 $ go test
2 PASS
3 ok      _/home/gopher/hello 0.020s
4 $
5 |
```

Creating modules

- The last line summarizes the overall package test.
- Because we are working outside \$GOPATH and also outside any module, the go command knows no import path for the current directory and makes up a fake one based on the directory name:
`_/home/gopher/hello.`
- Let's make the current directory the root of a module by using `go mod init` and then try `go test` again:

Creating modules

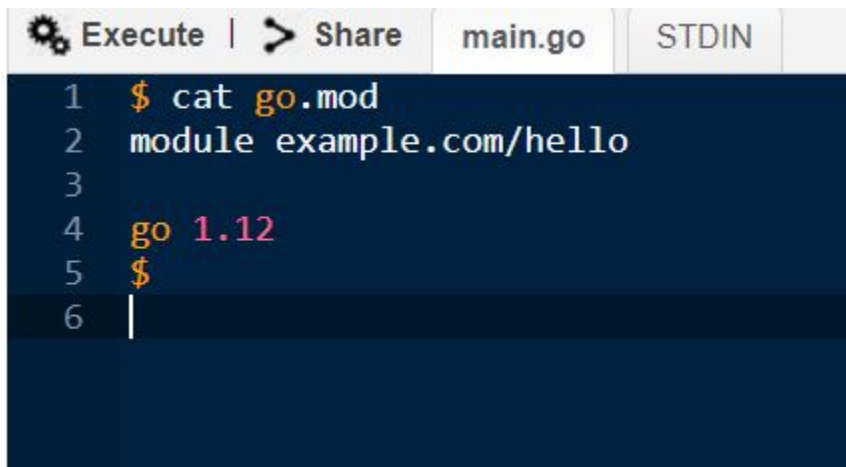


The screenshot shows a Go Playground interface with a dark blue background. At the top, there is a toolbar with a gear icon, the text "Execute |", a right-pointing arrow, the text "Share", and two tabs labeled "main.go" and "STDIN". Below the toolbar, a terminal window displays the following commands and output:

```
1 $ go mod init example.com/hello
2 go: creating new go.mod: module example.com/hello
3 $ go test
4 PASS
5 ok      example.com/hello    0.020s
6 $
7 |
```

Creating modules

The go mod init command wrote a go.mod file:

A screenshot of the Go Playground interface. At the top, there are tabs for 'Execute' (with a gear icon), 'Share' (with a share icon), 'main.go', and 'STDIN'. The 'main.go' tab is selected. Below the tabs is a dark blue code editor with a light blue line number margin on the left. The code in the editor is as follows:

```
1 $ cat go.mod
2 module example.com/hello
3
4 go 1.12
5 $
6 |
```

Creating modules

- The `go.mod` file only appears in the root of the module.
- Packages in subdirectories have import paths consisting of the module path plus the path to the subdirectory.
- For example, if we created a subdirectory `world`, we would not need to (nor want to) run `go mod init` there.
- The package would automatically be recognized as part of the `example.com/hello` module, with import path `example.com/hello/world`.

Using external modules

- Adding a Remote Module as a Dependency
 - Go modules are distributed from version control repositories, commonly Git repositories.
 - When you want to add a new module as a dependency to your own, you use the repository's path as a way to reference the module you'd like to use.

When Go sees the import path for these modules, it can infer where to find it remotely based on this repository path.

object-oriented constructs

- Object-oriented programming takes procedural programming a couple of steps further.
- Object-oriented programming started out as a new technique which allowed data to be divided into separated scopes called "objects".
- Only specific functions belonging to the same scope could access the same data.
- This is called encapsulation.

object-oriented constructs

- In the beginning objects were not called objects, they were just viewed upon as separate scopes.
- Later when dependencies were reduced and connections between functions and variables inside these scopes were viewed upon as isolated segments, the result gave birth to the concepts of "objects" and "object-oriented programming".

Methods and method receivers

- Go methods are similar to Go function with one difference, i.e, the method contains a receiver argument in it.
- With the help of the receiver argument, the method can access the properties of the receiver.
- Here, the receiver can be of struct type or non-struct type.
- When you create a method in your code the receiver and receiver type must be present in the same package.

Methods and method receivers

- And you are not allowed to create a method in which the receiver type is already defined in another package including inbuilt type like int, string, etc.
- If you try to do so, then the compiler will give an error.
- Syntax:

```
func(receiver_name Type)
method_name(parameter_list) (return_type) {
    // Code
}
```

Methods and method receivers

- Method with struct type receiver
 - In Go language, you are allowed to define a method whose receiver is of a struct type.
- Method with Non-Struct Type Receiver
 - In Go language, you are allowed to create a method with non-struct type receiver as long as the type and the method definitions are present in the same package.
 - If they present in different packages like `int`, `string`, etc, then the compiler will give an error because they are defined in different

Methods and method receivers

- Methods with Pointer Receiver
- In Go language, you are allowed to create a method with a pointer receiver.
- With the help of a pointer receiver, if a change is made in the method, it will reflect in the caller which is not possible with the value receiver methods.

Interfaces

- Go language interfaces are different from other languages.
- In Go language, the interface is a custom type that is used to specify a set of one or more method signatures and the interface is abstract, so you are not allowed to create an instance of the interface.
- But you are allowed to create a variable of an interface type and this variable can be assigned with a concrete type value that has the methods the interface requires.
- Or in other words, the interface is a collection of methods as well as it is a custom type.

Interfaces

- How to create an interface?
- In Go language, you can create an interface using the following syntax:

```
type interface_name interface{
```

```
// Method signatures
```

```
}
```

Interfaces

- How to implement interfaces?
- In the Go language, it is necessary to implement all the methods declared in the interface for implementing an interface.
- The go language interfaces are implemented implicitly.
- And it does not contain any specific keyword to implement an interface just like other languages.

Interfaces

- The zero value of the interface is nil.
- When an interface contains zero methods, such types of interface is known as the empty interface.
- So, all the types implement the empty interface.

Syntax:

```
interface{ }
```

Interfaces

- Interface Types:
 - static
 - dynamic
- A variable of the interface type containing the value of the Type which implements the interface, so the value of that Type is known as dynamic value and the type is the dynamic type.
- It is also known as concrete value and concrete type.

Interfaces

- Type Assertions: In Go language, type assertion is an operation applied to the value of the interface.
- Or in other words, type assertion is a process to extract the values of the interface.
- Syntax:
 - `a.(T)`
- Here, `a` is the value or the expression of the interface and `T` is the type also known as asserted type.

Interfaces

- The type assertion is used to check that the dynamic type of its operand will match the asserted type or not.
- If the T is of concrete type, then the type assertion checks the given dynamic type of a is equal to the T, here if the checking proceeds successfully, then the type assertion returns the dynamic value of a.
- Or if the checking fails, then the operation will panic.
- If the T is of an interface type, then the type assertion checks the given dynamic type of a satisfies T, here if the checking proceeds successfully, then the dynamic value is not extracted.

type assertions

- Type assertions in Golang provide access to the exact type of variable of an interface.
- If already the data type is present in the interface, then it will retrieve the actual data type value held by the interface.
- A type assertion takes an interface value and extracts from it a value of the specified explicit type.
- Basically, it is used to remove the ambiguity from the interface variables.

type assertions

- Syntax:
- `t := value.(typeName)`
- where `value` is a variable whose type must be an interface, `typeName` is the concrete type we want to check and underlying `typeName` value is assigned to variable `t`.

Constructors

- There are no default constructors in Go, but you can declare methods for any type.
- You could make it a habit to declare a method called "Init".
- Go doesn't support constructors, but constructor-like factory functions are easy to implement:

Constructors



Execute



Share

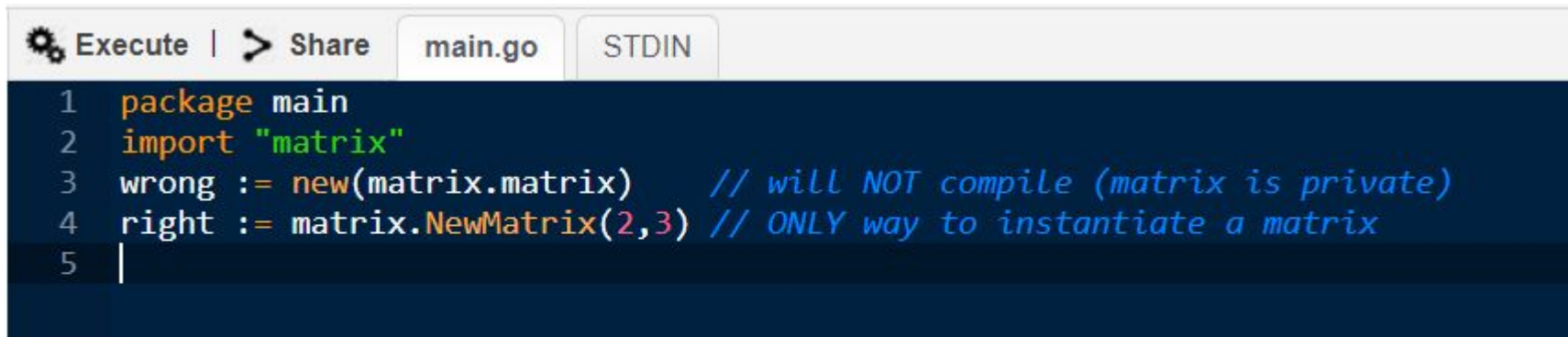
main.go

STDIN

```
1 package matrix
2 function NewMatrix(rows, cols int) *matrix {
3     m := new(matrix)
4     m.rows = rows
5     m.cols = cols
6     m.elems = make([]float, rows*cols)
7     return m
8 }
9
10 |
```

Constructors

- To prevent users from instantiating uninitialized objects, the struct can be made private.



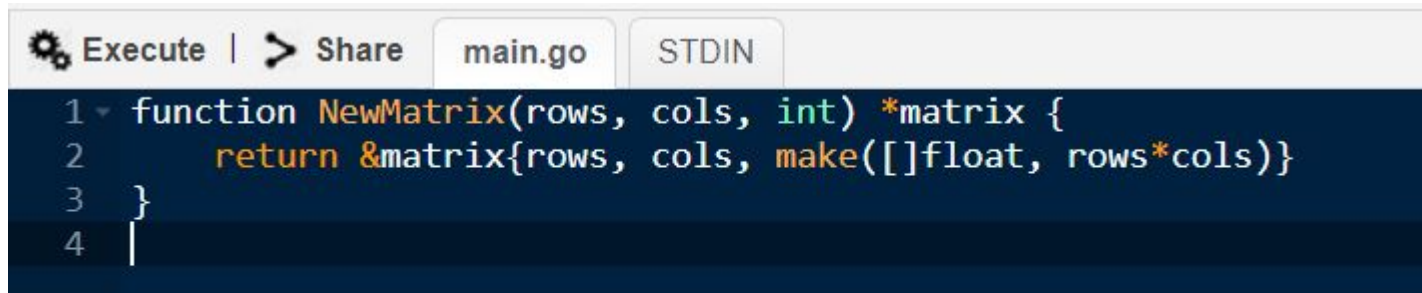
```
Execute | Share main.go STDIN
1 package main
2 import "matrix"
3 wrong := new(matrix.matrix) // will NOT compile (matrix is private)
4 right := matrix.NewMatrix(2,3) // ONLY way to instantiate a matrix
5 |
```

Constructors

- Initializers
- Go aims to prevent unnecessary typing.
- Oftentimes Go code is shorter and easier to read than object-oriented languages.
- `matrix := NewMatrix(10, 10)`
- `pair := &Pair{"one", 1}`

Constructors

- Additionally, Go "constructors" can be written succinctly using initializers within a factory function:

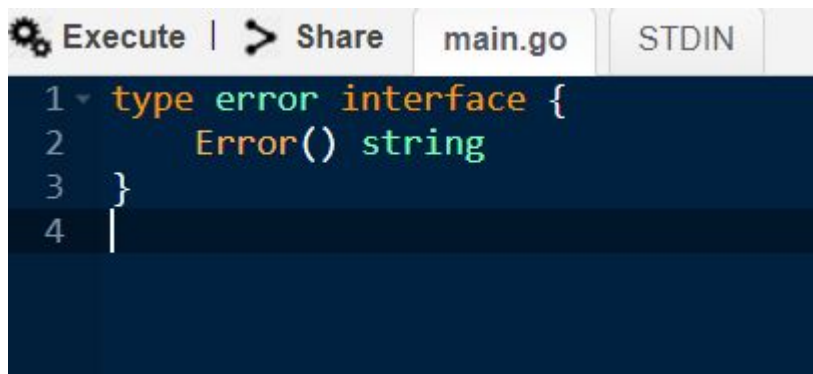


The screenshot shows a code editor interface with a toolbar at the top containing a gear icon, the text "Execute", a share icon, the text "Share", and two tabs labeled "main.go" and "STDIN". The code is written in a dark-themed editor with syntax highlighting. It defines a function named "NewMatrix" that takes three arguments: "rows", "cols", and "int", and returns a pointer to a "matrix" struct. The struct is initialized with "rows", "cols", and a slice of floats created by "make([]float, rows*cols)".

```
1 function NewMatrix(rows, cols, int) *matrix {  
2     return &matrix{rows, cols, make([]float, rows*cols)}  
3 }  
4 |
```

Constructors

- where the predeclared type `error` is defined as an interface:



The screenshot shows a code editor with a dark blue background. At the top, there are tabs for 'Execute', 'Share', 'main.go', and 'STDIN'. The code is written in Go and defines the `error` interface. The lines are numbered 1 through 4 on the left side of the editor.

```
1 type error interface {  
2     Error() string  
3 }  
4 |
```