Learning Objectives

Learners will be able to...

- **use Go pointers**
- **implement interfaces**

info

Make Sure You Know

Python or other modern programming language.

Pointers

A pointer is a variable that stores the memory address of an object. For larger data structures, the pointer indicates the address of the first cell in which the object is located.

---

info

Pointer operations

There are two main operations for working with pointers: assignment and dereference.

Assignment assigns a pointer to a certain address.

Dereferencing accesses the value referenced by a pointer.

A pointer can also have a `nil` address indicating that the pointer does not refer to any memory location.

Pointers in Go

In Go, a pointer is defined in the same way as a regular variable, but the type is preceded by an asterisk `*`:

```
var strPtr *string
var intPtr *int
```

The pointer is declared with the type of object it points to. In the above example, `strPtr` points to a `string` variable, and `intPtr` points to an `int` variable.

Try out the code below by pasting it into the file on the left and clicking the Run button below:

```
func main() {
    var name string = "Michael"
    var ptr *string
    ptr = &name
    fmt.Println(ptr)
}
```

As seen in the example above:
* Use & to get the address of a variable.
* Using fmt.Println() you can print the address of a variable, represented as a hexadecimal value.

To **dereference** a pointer, or retrieve the value it is pointing to, put * before the pointer.
Try derefencing the pointer using *

```
fmt.Println(*ptr)
```

To declare and assign pointers for existing variables, you can use the shorthand form:

```
var name string = "Michael"
ptr := &name
```

`nil` Pointers

When getting a `nil` or zero address pointer value, a `runtime error: invalid memory address or nil pointer dereference` error will be raised.

To handle this, pointers should be checked in cases where the adress may turn out to be `nil`:

```
var name string = "Michael"
var ptr *string
ptr = &name
if ptr != nil { // We check whether the address is null.
    fmt.Println(*ptr) // Michael
}
ptr = nil
fmt.Println(*ptr) // panic: runtime error: invalid memory
        address or nil pointer dereference
```

Using pointers in functions

Pointers can be used as function parameters. Using a pointer means we can also change the value of the variable to which the pointer refers.

```
func main() {
    var name string = "Michael"
    var ptr *string
    ptr = &name
    fmt.Println(name)
    changeName(ptr, "Jones")
    fmt.Println(name)
}

func changeName(ptr *string, name string) {
    *ptr = name
}
```

A pointer can also be returned as a result of executing a function.

```
func getName() *string {
    name := "Jack"
    return &name
}
```

Pointers can be used as optional arguments

Consider the following example:

```
func main() {
  var xCoord int = 5
  var yCoord int = 7
  xPtr := &xCoord
  yPtr := &yCoord
  fmt.Printf("(%d,%d)\n", xCoord, yCoord)
  incrementPosition(xPtr, yPtr)
  fmt.Printf("(%d,%d)\n", xCoord, yCoord)
}

func incrementPosition(x, y *int) {
  *x = *x+1
  *y = *y+1
}
```

But what if we want to change only the value of the `x` coordinate?
In the case of Python, we could simply set `y` to `None`. But in the case of Go, if we pass `nil` as an argument, we will get a compilation error: `cannot use nil as int value in argument to updatePosition`.

In this case, you can pass pointers to the function and check it for `nil`. As a result we get the following:

```
func main() {
  var xCoord int = 5
  var yCoord int = 7
  xPtr := &xCoord
  fmt.Printf("(%d,%d)\n", xCoord, yCoord)
  incrementPosition(xPtr, nil)
  fmt.Printf("(%d,%d)\n", xCoord, yCoord)
}

func incrementPosition(x, y *int) {
  if x != nil {
      *x = *x+1
  }
  if y != nil {
      *y = *y+1
  }
}
```

`new` function

The `new` function allows you to create an unnamed object.

We pass the type being created to the `new` function, and it returns a pointer to the created object:

```
func main() {
    ptrName := new(string)
    fmt.Println(*ptrName) // prints empty line - default value
of `string`
    *ptrName = "Michael"
    fmt.Println(*ptrName)
}
```

Note that while the memory is allocated and can be accessed by the returned pointer, the object is not named like a variable.

Structs and Interfaces

Structs and interfaces are ways to group variables and functions in Go.

definition

Structures

A struct is a user-defined type that holds a group of fields.

The syntax is:

```
type structureName struct {
    varName type
}
```

For example, let's take two structures `book` and `magazine`.

```
type book struct {
    title string
    isbn  int
}

type magazine struct {
    title string
    issn  int
}
```

definition

Interfaces

Interfaces are a set of method signatures.

The syntax is:

```
type interfaceName interface {
    methodSignature()
}
```

The interface defines the functionality of the behavior but does not implement it.

For example, let's create the `viewer` interface and define the `getTitle` method in it, which will allow us to get the title of the printed publication.

```
type viewer interface {
    getTitle() string
}
```

Remember that an interface is an abstraction, and we cannot create an interface object. For example, this code will throw an error:

```
v := viewer{}
```

Implementing interfaces

Interfaces in Go are implemented implicitly.

In order for our `structs` to correspond to this interface, it is necessary to implement **all** the interface methods for each data type:

```
func (b book) getTitle() string {
    return b.title
}

func (m magazine) getTitle() string {
    return m.title
}
```

Next we can execute add the following code:

```
myBook := book{title: "Clean Code", isbn: 123}
myMagazine := magazine{title: "National Geographic", issn: 456}

var myViewer viewer

myViewer = myBook
myViewer = myMagazine
```

Let's add the `getStandardNumber` method to our `viewer` interface.

```
type viewer interface {
    getTitle() string
    getStandardNumber() int
}
```

And for now we will implement it only for the `book` type.

```
func (b book) getStandardNumber() int {
    return b.isbn
}
```

Next we can execute the code again and it will become clear that now the `magazine` type does not correspond to our interface since it does not implement all of its methods.

Note that the last line will throw an error:

```
cannot use myMagazine (variable of type magazine)
as viewer value in assignment:
magazine does not implement viewer
(missing method getStandardNumber)
```

▼ **Click to see the above example code all put together**

```
package main

import "fmt"

type viewer interface {
    getTitle() string
    getStandartNumber() int
}

type book struct {
    title string
    isbn  int
}

func (b book) getTitle() string {
    return b.title
}

func (b book) getStandartNumber() int {
    return b.isbn
}

type magazine struct {
    title string
    issn  int
}

func (m magazine) getTitle() string {
    return m.title
}

func main() {
    myBook := book{title: "Clean Code", isbn: 123}
    myMagazine := magazine{title: "National Geographic",
issn: 456}

    var myViewer viewer

    myViewer = myBook
    myViewer = myMagazine // here we will get an error

    fmt.Println(myViewer.getTitle())
}
```

Handling `nil`

Similar to pointers, `nil` interfaces should be handled:

```
func (b *book) getTitle() string {
    if b == nil {
        fmt.Println("book is nil") // We will handle the error
accordingly.
        return ""
    }
    return b.title
}
```

Empty interfaces

There are **empty interfaces** or interfaces without methods.

Since every type implements at least zero methods, all data types implement an empty interface.

Empty interfaces are used when it is necessary to process values of an unknown type.

For example, an empty interface will correspond to the types `book`, `magazine`, `int`:

```
func main() {

    var myVar interface{}

    myBook := book{title: "Clean Code", isbn: 123}
    myMagazine := magazine{title: "National Geographic", issn:
456}
    number := 12

    myVar = myBook
    myVar = myMagazine
    myVar = number

    fmt.Println(myVar)
}
```

As you can see, the `fmt.Println` function accepts values of type `interface{}`.

It is convenient to use empty interfaces when parsing `json`.

Types

Consider our previous example:

```
type book struct {
    title string
    isbn  int
}

func (b book) getTitle() string {
    return b.title
}

type magazine struct {
    title string
    issn  int
}

func (m magazine) getTitle() string {
    return m.title
}
```

Next, let's create the `myBook` and `myMagazine` objects and print their title:

```
func main() {
    myBook := book{title: "Clean Code", isbn: 123}
    myMagazine := magazine{title: "National Geographic", issn:
        456}

    printBookTitle(myBook)
    printMagazineTitle(myMagazine)
}

func printBookTitle(book book) {
    fmt.Println(book.getTitle())
}

func printMagazineTitle(magazine magazine) {
    fmt.Println(magazine.getTitle())
}
```

As you can see, the `printBookTitle` and `printMagazineTitle` functions duplicate each other. Now, as in previous examples, we will add an interface:

```
type viewer interface {
    getTitle() string
}
```

Then we update the code to:

```
func main() {
    myBook := book{title: "Clean Code", isbn: 123}
    myMagazine := magazine{title: "National Geographic", issn:
456}

    printTitle(myBook)
    printTitle(myMagazine)
}

func printTitle(printedMatter viewer) {
    fmt.Println(printedMatter.getTitle())
}
```

Since both our `book` and `magazine` structures correspond to the `viewer` interface, we can pass them as an argument to our `printTitle` function, which takes values of type `viewer`.

Type assertions

A **type assertion** provides access to the underlying concrete value of an interface's value.

In general it will look like:

```
t := i.(T)
```

If the operation is successful, we will receive a variable of type `T`, otherwise we will get panic. In order to avoid panic, you can use the following construction:

```
t, ok := i.(T)
```

Now if the interface contains `T` the variable `ok` will contain `true`, and `t` will contain a value of type `T`.

You can try this out with the `book` struct:

```
package main

import (
    "fmt"
    "reflect"
)

type book struct {
    title string
    isbn  int
}


func main() {
    var iBook interface{} = book{title: "Clean Code", isbn: 123}
    myBook, ok := iBook.(book)
    if ok {
        fmt.Println(reflect.TypeOf(myBook))
        fmt.Println(myBook)
    } else {
        fmt.Println("`iBook` interface does not contain `book`")
    }
}
```

Type switches

A **type switch** is a construct that allows us to perform certain actions based on the type of the values contained in the interface.

A declaration in a switch has the same syntax as an assertion, but instead of a type, the keyword `type` is specified:

```
func main() {
    var iBook interface{} = book{title: "Clean Code", isbn: 123}
    switch printedMatterType := iBook.(type) {
    case book:
        fmt.Println("is book")
        fmt.Println(printedMatterType)
    case magazine:
        fmt.Println("is magazine")
        fmt.Println(printedMatterType)
    default:
        fmt.Println("no matches")
    }
}
```

Multiple and Nested interfaces

A data type can have both its own methods and implementation of other interfaces. This also means **a data type can implement multiple interfaces.**

Let's add a new interface `editor` and implement it:

```
type editor interface {
    setTitle(string)
}

func (b *book) setTitle(title string) {
    b.title = title
}
```

This way we can expand the functionality of the behavior of our objects.

Here is the full example to try:

```
package main

import  "fmt"

type viewer interface {
    getTitle() string
}

type editor interface {
    setTitle(string)
}

type book struct {
    title string
    isbn  int
}

func (b book) getTitle() string {
    return b.title
}

func (b *book) setTitle(title string) {
    b.title = title
}

func main() {
    myBook := book{title: "Clean Code", isbn: 123}
    fmt.Println(myBook.getTitle())
    myBook.setTitle("Clean Architecture")
    fmt.Println(myBook.getTitle())
}
```

Nested interfaces

Interfaces can contain other interfaces. In order for a data type to correspond to an interface with nested interfaces, it must implement all of them.

```
package main

import "fmt"

type viewer interface {
    getTitle() string
}

type editor interface {
    setTitle(string)
}

type viewerEditor interface {
    viewer
    editor
}

type book struct {
    title string
    isbn  int
}

func (b book) getTitle() string {
    return b.title
}

func (b *book) setTitle(title string) {
    b.title = title
}

func main() {
    var myBook viewerEditor
    myBook = &book{title: "Clean Code", isbn: 123}
    fmt.Println(myBook.getTitle())
    myBook.setTitle("Clean Architecture")
    fmt.Println(myBook.getTitle())
}
```