Pointers & errors

You can find all the code for this chapter here

We learned about structs in the last section which let us capture a number of values related around a concept.

At some point you may wish to use structs to manage state, exposing methods to let users change the state in a way that you can control.

Fintech loves Go and uhhh bitcoins? So let's show what an amazing banking system we can make.

Let's make a `Wallet` struct which lets us deposit `Bitcoin`.

Write the test first

```
1  func TestWallet(t *testing.T) {
2
3          wallet := Wallet{}
4
5          wallet.Deposit(10)
6
7          got := wallet.Balance()
8          want := 10
9
10         if got != want {
11                 t.Errorf("got %d want %d", got, want)
12         }
13 }
```

In the previous example we accessed fields directly with the field name, however in our *very secure wallet* we don't want to expose our inner state to the rest of the world. We want to control access via methods.

Try to run the test

`./wallet_test.go:7:12: undefined: Wallet`

Write the minimal amount of code for the test to run and check the failing test output

The compiler doesn't know what a `Wallet` is so let's tell it.

```
type Wallet struct{}
```

Now we've made our wallet, try and run the test again

```
./wallet_test.go:9:8: wallet.Deposit undefined (type Wallet has no field or method De
./wallet_test.go:11:15: wallet.Balance undefined (type Wallet has no field or method
```

We need to define these methods.

Remember to only do enough to make the tests run. We need to make sure our test fails correctly with a clear error message.

```
func (w Wallet) Deposit(amount int) {

}

func (w Wallet) Balance() int {
        return 0
}
```

If this syntax is unfamiliar go back and read the structs section.

The tests should now compile and run

`wallet_test.go:15: got 0 want 10`

Write enough code to make it pass

We will need some kind of *balance* variable in our struct to store the state

```
type Wallet struct {
        balance int
}
```

In Go if a symbol (variables, types, functions et al) starts with a lowercase symbol then it is private *outside the package it's defined in*.

In our case we want our methods to be able to manipulate this value, but no one else.

Remember we can access the internal `balance` field in the struct using the "receiver" variable.

```
func (w Wallet) Deposit(amount int) {
        w.balance += amount
}

func (w Wallet) Balance() int {
        return w.balance
}
```

With our career in fintech secured, run the test suite and bask in the passing test

`wallet_test.go:15: got 0 want 10`

That's not quite right

Well this is confusing, our code looks like it should work. We add the new amount onto our balance and then the balance method should return the current state of it.

In Go, **when you call a function or a method the arguments are *copied***.

When calling `func (w Wallet) Deposit(amount int)` the `w` is a copy of whatever we called the method from.

Without getting too computer-sciency, when you create a value - like a wallet, it is stored

Experiment by adding some prints to your code

```
func TestWallet(t *testing.T) {

        wallet := Wallet{}

        wallet.Deposit(10)

        got := wallet.Balance()

        fmt.Printf("address of balance in test is %v \n", &wallet.balance)

        want := 10

        if got != want {
                t.Errorf("got %d want %d", got, want)
        }
}
```

```
func (w Wallet) Deposit(amount int) {
        fmt.Printf("address of balance in Deposit is %v \n", &w.balance)
        w.balance += amount
}
```

The escape character prints a new line after outputting the memory address. We get the pointer (memory address) of something by placing an `&` character at the beginning of the symbol.

Now re-run the test

```
address of balance in Deposit is 0xc420012268
address of balance in test is 0xc420012260
```

You can see that the addresses of the two balances are different. So when we change the value of the balance inside the code, we are working on a copy of what came from the test. Therefore the balance in the test is unchanged.

We can fix this with *pointers*. Pointers let us *point* to some values and then let us change them. So rather than taking a copy of the whole Wallet, we instead take a pointer to that wallet so that we can

change the original values within it.

```
func (w *Wallet) Deposit(amount int) {
        w.balance += amount
}

func (w *Wallet) Balance() int {
        return w.balance
}
```

The difference is the receiver type is `*Wallet` rather than `Wallet` which you can read as "a pointer to a wallet".

Try and re-run the tests and they should pass.

Now you might wonder, why did they pass? We didn't dereference the pointer in the function, like so:

```
func (w *Wallet) Balance() int {
        return (*w).balance
}
```

and seemingly addressed the object directly. In fact, the code above using `(*w)` is absolutely valid. However, the makers of Go deemed this notation cumbersome, so the language permits us to write `w.balance`, without an explicit dereference. These pointers to structs even have their own name: *struct pointers* and they are automatically dereferenced.

Technically you do not need to change `Balance` to use a pointer receiver as taking a copy of the balance is fine. However, by convention you should keep your method receiver types the same for consistency.

Refactor

We said we were making a Bitcoin wallet but we have not mentioned them so far. We've been using `int` because they're a good type for counting things!

It seems a bit overkill to create a `struct` for this. `int` is fine in terms of the way it works but it's

not descriptive.

Go lets you create new types from existing ones.

The syntax is `type MyName OriginalType`

```
type Bitcoin int

type Wallet struct {
        balance Bitcoin
}

func (w *Wallet) Deposit(amount Bitcoin) {
        w.balance += amount
}

func (w *Wallet) Balance() Bitcoin {
        return w.balance
}
```

```
func TestWallet(t *testing.T) {

        wallet := Wallet{}

        wallet.Deposit(Bitcoin(10))

        got := wallet.Balance()

        want := Bitcoin(10)

        if got != want {
                t.Errorf("got %d want %d", got, want)
        }
}
```

To make `Bitcoin` you just use the syntax `Bitcoin(999)`.

By doing this we're making a new type and we can declare *methods* on them. This can be very useful when you want to add some domain specific functionality on top of existing types.

Let's implement Stringer on Bitcoin

```
type Stringer interface {
        String() string
}
```

This interface is defined in the `fmt` package and lets you define how your type is printed when used with the `%s` format string in prints.

```
func (b Bitcoin) String() string {
        return fmt.Sprintf("%d BTC", b)
}
```

As you can see, the syntax for creating a method on a type declaration is the same as it is on a struct.

Next we need to update our test format strings so they will use `String()` instead.

```
        if got != want {
                t.Errorf("got %s want %s", got, want)
        }
```

To see this in action, deliberately break the test so we can see it

`wallet_test.go:18: got 10 BTC want 20 BTC`

This makes it clearer what's going on in our test.

The next requirement is for a `Withdraw` function.

Write the test first

Pretty much the opposite of `Deposit()`

```
func TestWallet(t *testing.T) {

        t.Run("Deposit", func(t *testing.T) {
```

```
4                 wallet := Wallet{}
5
6                 wallet.Deposit(Bitcoin(10))
7
8                 got := wallet.Balance()
9
10                want := Bitcoin(10)
11
12                if got != want {
13                        t.Errorf("got %s want %s", got, want)
14                }
15        })
16
17        t.Run("Withdraw", func(t *testing.T) {
18                wallet := Wallet{balance: Bitcoin(20)}
19
20                wallet.Withdraw(Bitcoin(10))
21
22                got := wallet.Balance()
23
24                want := Bitcoin(10)
25
26                if got != want {
27                        t.Errorf("got %s want %s", got, want)
28                }
29        })
30 }
```

Try to run the test

`./wallet_test.go:26:9: wallet.Withdraw undefined (type Wallet has no field or method Withdraw)`

Write the minimal amount of code for the test to run and check the failing test output

```
func (w *Wallet) Withdraw(amount Bitcoin) {

}
```

```
wallet_test.go:33: got 20 BTC want 10 BTC
```

Write enough code to make it pass

```
func (w *Wallet) Withdraw(amount Bitcoin) {
        w.balance -= amount
}
```

Refactor

There's some duplication in our tests, lets refactor that out.

```
func TestWallet(t *testing.T) {

        assertBalance := func(t testing.TB, wallet Wallet, want Bitcoin) {
                t.Helper()
                got := wallet.Balance()

                if got != want {
                        t.Errorf("got %s want %s", got, want)
                }
        }

        t.Run("Deposit", func(t *testing.T) {
                wallet := Wallet{}
                wallet.Deposit(Bitcoin(10))
                assertBalance(t, wallet, Bitcoin(10))
        })

        t.Run("Withdraw", func(t *testing.T) {
                wallet := Wallet{balance: Bitcoin(20)}
```

```
                wallet.Withdraw(Bitcoin(10))
                assertBalance(t, wallet, Bitcoin(10))

        })

}
```

What should happen if you try to `Withdraw` more than is left in the account? For now, our requirement is to assume there is not an overdraft facility.

How do we signal a problem when using `Withdraw`?

In Go, if you want to indicate an error it is idiomatic for your function to return an `err` for the caller to check and act on.

Let's try this out in a test.

Write the test first

```
t.Run("Withdraw insufficient funds", func(t *testing.T) {
        startingBalance := Bitcoin(20)
        wallet := Wallet{startingBalance}
        err := wallet.Withdraw(Bitcoin(100))

        assertBalance(t, wallet, startingBalance)

        if err == nil {
                t.Error("wanted an error but didn't get one")
        }
})
```

We want `Withdraw` to return an error *if* you try to take out more than you have and the balance should stay the same.

We then check an error has returned by failing the test if it is `nil`.

`nil` is synonymous with `null` from other programming languages. Errors can be `nil` because the return type of `Withdraw` will be `error`, which is an interface. If you see a function that takes arguments or returns values that are interfaces, they can be nillable.

Like `null` if you try to access a value that is `nil` it will throw a **runtime panic**. This is bad! You should make sure that you check for nils.

Try and run the test

`./wallet_test.go:31:25: wallet.Withdraw(Bitcoin(100)) used as value`

The wording is perhaps a little unclear, but our previous intent with `Withdraw` was just to call it, it will never return a value. To make this compile we will need to change it so it has a return type.

Write the minimal amount of code for the test to run and check the failing test output

```
1 func (w *Wallet) Withdraw(amount Bitcoin) error {
2         w.balance -= amount
3         return nil
4 }
```

Again, it is very important to just write enough code to satisfy the compiler. We correct our `Withdraw` method to return `error` and for now we have to return *something* so let's just return `nil`.

Write enough code to make it pass

```
1 func (w *Wallet) Withdraw(amount Bitcoin) error {
2
3         if amount > w.balance {
4                 return errors.New("oh no")
5         }
6
7         w.balance -= amount
8         return nil
```

```
}
```

Remember to import `errors` into your code.

`errors.New` creates a new `error` with a message of your choosing.

Refactor

Let's make a quick test helper for our error check to improve the test's readability

```
assertError := func(t testing.TB, err error) {
        t.Helper()
        if err == nil {
                t.Error("wanted an error but didn't get one")
        }
}
```

And in our test

```
t.Run("Withdraw insufficient funds", func(t *testing.T) {
        startingBalance := Bitcoin(20)
        wallet := Wallet{startingBalance}
        err := wallet.Withdraw(Bitcoin(100))

        assertError(t, err)
        assertBalance(t, wallet, startingBalance)
})
```

Hopefully when returning an error of "oh no" you were thinking that we *might* iterate on that because it doesn't seem that useful to return.

Assuming that the error ultimately gets returned to the user, let's update our test to assert on some kind of error message rather than just the existence of an error.

Write the test first

Update our helper for a `string` to compare against.

```
assertError := func(t testing.TB, got error, want string) {
        t.Helper()
        if got == nil {
                t.Fatal("didn't get an error but wanted one")
        }

        if got.Error() != want {
                t.Errorf("got %q, want %q", got, want)
        }
}
```

And then update the caller

```
t.Run("Withdraw insufficient funds", func(t *testing.T) {
        startingBalance := Bitcoin(20)
        wallet := Wallet{startingBalance}
        err := wallet.Withdraw(Bitcoin(100))

        assertError(t, err, "cannot withdraw, insufficient funds")
        assertBalance(t, wallet, startingBalance)
})
```

We've introduced `t.Fatal` which will stop the test if it is called. This is because we don't want to make any more assertions on the error returned if there isn't one around. Without this the test would carry on to the next step and panic because of a nil pointer.

Try to run the test

```
wallet_test.go:61: got err 'oh no' want 'cannot withdraw, insufficient funds'
```

Write enough code to make it pass

```go
func (w *Wallet) Withdraw(amount Bitcoin) error {

        if amount > w.balance {
                return errors.New("cannot withdraw, insufficient funds")
        }

        w.balance -= amount
        return nil
}
```

Refactor

We have duplication of the error message in both the test code and the `Withdraw` code.

It would be really annoying for the test to fail if someone wanted to re-word the error and it's just too much detail for our test. We don't *really* care what the exact wording is, just that some kind of meaningful error around withdrawing is returned given a certain condition.

In Go, errors are values, so we can refactor it out into a variable and have a single source of truth for it.

```go
var ErrInsufficientFunds = errors.New("cannot withdraw, insufficient funds")

func (w *Wallet) Withdraw(amount Bitcoin) error {

        if amount > w.balance {
                return ErrInsufficientFunds
        }

        w.balance -= amount
        return nil
}
```

The `var` keyword allows us to define values global to the package.

This is a positive change in itself because now our `Withdraw` function looks very clear.

Next we can refactor our test code to use this value instead of specific strings.

```
func TestWallet(t *testing.T) {

        t.Run("Deposit", func(t *testing.T) {
                wallet := Wallet{}
                wallet.Deposit(Bitcoin(10))
                assertBalance(t, wallet, Bitcoin(10))
        })

        t.Run("Withdraw with funds", func(t *testing.T) {
                wallet := Wallet{Bitcoin(20)}
                wallet.Withdraw(Bitcoin(10))
                assertBalance(t, wallet, Bitcoin(10))
        })

        t.Run("Withdraw insufficient funds", func(t *testing.T) {
                wallet := Wallet{Bitcoin(20)}
                err := wallet.Withdraw(Bitcoin(100))

                assertError(t, err, ErrInsufficientFunds)
                assertBalance(t, wallet, Bitcoin(20))
        })
}

func assertBalance(t testing.TB, wallet Wallet, want Bitcoin) {
        t.Helper()
        got := wallet.Balance()

        if got != want {
                t.Errorf("got %q want %q", got, want)
        }
}

func assertError(t testing.TB, got, want error) {
        t.Helper()
        if got == nil {
                t.Fatal("didn't get an error but wanted one")
        }

        if got != want {
                t.Errorf("got %q, want %q", got, want)
        }
```

```
42 }
```

And now the test is easier to follow too.

I have moved the helpers out of the main test function just so when someone opens up a file they can start reading our assertions first, rather than some helpers.

Another useful property of tests is that they help us understand the *real* usage of our code so we can make sympathetic code. We can see here that a developer can simply call our code and do an equals check to `ErrInsufficientFunds` and act accordingly.

Unchecked errors

Whilst the Go compiler helps you a lot, sometimes there are things you can still miss and error handling can sometimes be tricky.

There is one scenario we have not tested. To find it, run the following in a terminal to install `errcheck`, one of many linters available for Go.

`go get -u github.com/kisielk/errcheck` ()

```
1 in Golang 1.17 installing by 'go get' deprecated. Need to use combination:
2 `go get github.com/kisielk/errcheck`
3 `go install github.com/kisielk/errcheck`
```

Then, inside the directory with your code run `errcheck .`

You should get something like

`wallet_test.go:17:18: wallet.Withdraw(Bitcoin(10))`

What this is telling us is that we have not checked the error being returned on that line of code. That line of code on my computer corresponds to our normal withdraw scenario because we have not checked that if the `Withdraw` is successful that an error is *not* returned.

Here is the final test code that accounts for this.

```
1 func TestWallet(t *testing.T) {
2
3         t.Run("Deposit", func(t *testing.T) {
4                 wallet := Wallet{}
```

```
                wallet.Deposit(Bitcoin(10))

                assertBalance(t, wallet, Bitcoin(10))
        })

        t.Run("Withdraw with funds", func(t *testing.T) {
                wallet := Wallet{Bitcoin(20)}
                err := wallet.Withdraw(Bitcoin(10))

                assertNoError(t, err)
                assertBalance(t, wallet, Bitcoin(10))
        })

        t.Run("Withdraw insufficient funds", func(t *testing.T) {
                wallet := Wallet{Bitcoin(20)}
                err := wallet.Withdraw(Bitcoin(100))

                assertError(t, err, ErrInsufficientFunds)
                assertBalance(t, wallet, Bitcoin(20))
        })
}

func assertBalance(t testing.TB, wallet Wallet, want Bitcoin) {
        t.Helper()
        got := wallet.Balance()

        if got != want {
                t.Errorf("got %s want %s", got, want)
        }
}

func assertNoError(t testing.TB, got error) {
        t.Helper()
        if got != nil {
                t.Fatal("got an error but didn't want one")
        }
}

func assertError(t testing.TB, got error, want error) {
        t.Helper()
        if got == nil {
                t.Fatal("didn't get an error but wanted one")
        }

```

```
49          if got != want {
50                  t.Errorf("got %s, want %s", got, want)
51          }
52
```

Wrapping up

Pointers

- Go copies values when you pass them to functions/methods, so if you're writing a function that needs to mutate state you'll need it to take a pointer to the thing you want to change.
- The fact that Go takes a copy of values is useful a lot of the time but sometimes you won't want your system to make a copy of something, in which case you need to pass a reference. Examples include referencing very large data structures or things where only one instance is necessary (like database connection pools).

nil

- Pointers can be nil
- When a function returns a pointer to something, you need to make sure you check if it's nil or you might raise a runtime exception - the compiler won't help you here.
- Useful for when you want to describe a value that could be missing

Errors

- Errors are the way to signify failure when calling a function/method.
- By listening to our tests we concluded that checking for a string in an error would result in a flaky test. So we refactored our implementation to use a meaningful value instead and this resulted in easier to test code and concluded this would be easier for users of our API too.
- This is not the end of the story with error handling, you can do more sophisticated things but this is just an intro. Later sections will cover more strategies.
- Don't just check errors, handle them gracefully

Create new types from existing ones

- Useful for adding more domain specific meaning to values
- Can let you implement interfaces

Pointers and errors are a big part of writing Go that you need to get comfortable with. Thankfully the compiler will *usually* help you out if you do something wrong, just take your time and read the error.