[this slide exists because the timer in keynote only starts once you advance to the next slide]

Hello, welcome to Dot Go.

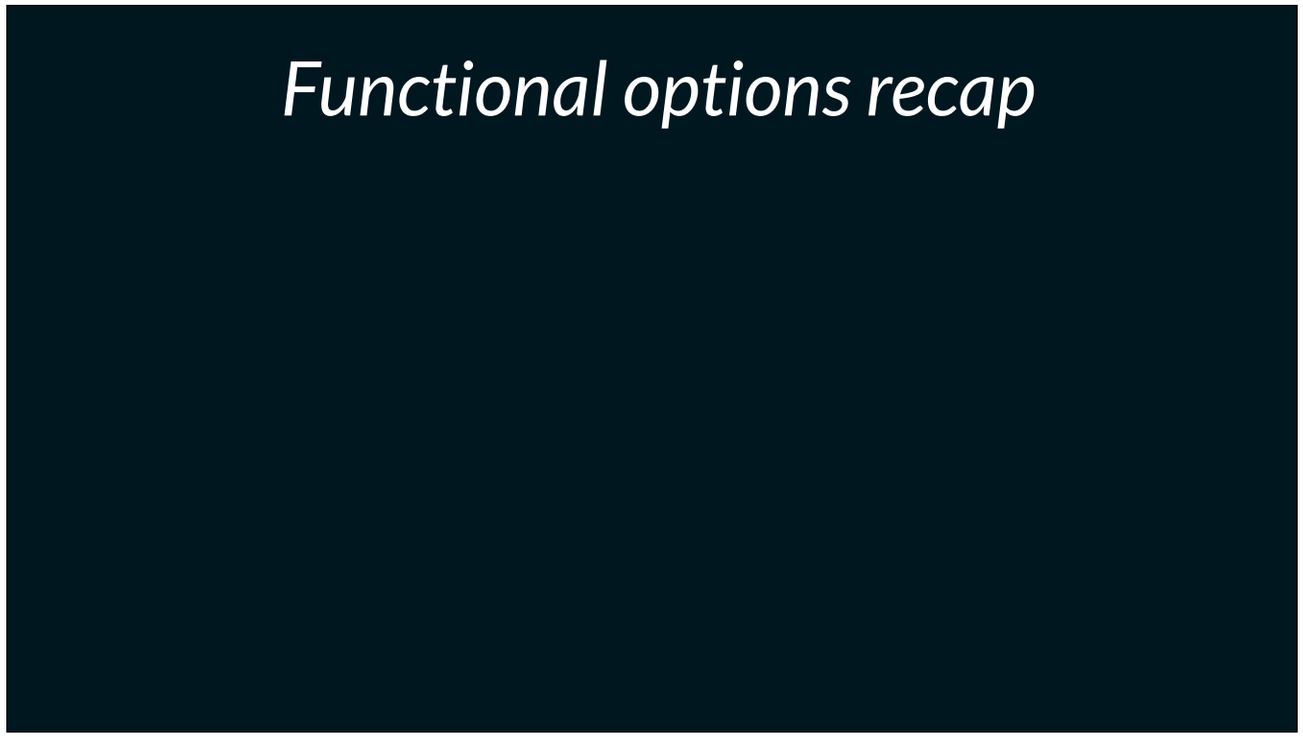
Two years ago I stood on a stage, not unlike this one, and told you my opinion for how configuration options should be handled in Go.

The cornerstone of my presentation was Rob Pike's blog post, Self-referential functions and the design of options.

In the last two years it has been wonderful to watch this idea mature from Rob's original blog post, to the gRPC project, who in my opinion have continued to evolve this design pattern into its best form to date.

But, when I was talking to Gophers at a conference in London a few months ago, several of them expressed a concern that while *they* understood the notion of a function that returns a function—this is the technique that powers functional options—they were worried that other Go programmers—I suspect they meant less experienced Go programmers—would not be able to understand this style of programming.

And this made me a bit sad, because I consider Go's support of first class functions to be a gift, and something that we should all be able to take advantage of. So I'm here today to show you, that you do not need to fear first class functions.



01:30

To begin, I'll very quickly recap the functional options pattern

[click] We start with some options, expressed as functions which take a pointer to a structure to configure.

[click] We pass those functions to a constructor, and inside the body of that constructor each option function is invoked in order, passing in a reference to the Config value.

[click] finally we call NewTerrain with the options we want. And away we go

Ok, everyone should be familiar with this pattern.

Where I believe the confusion comes from, is when you need an option function which take a parameter.

Functional options recap

```
type Config struct{}

func WithReticulatedSplines(c *Config) { ... }
```

01:30

To begin, I'll very quickly recap the functional options pattern

[click] We start with some options, expressed as functions which take a pointer to a structure to configure.

[click] We pass those functions to a constructor, and inside the body of that constructor each option function is invoked in order, passing in a reference to the Config value.

[click] finally we call NewTerrain with the options we want. And away we go

Ok, everyone should be familiar with this pattern.

Where I believe the confusion comes from, is when you need an option function which take a parameter.

Functional options recap

```
type Config struct{}

func WithReticulatedSplines(c *Config) { ... }

type Terrain struct {
        config Config
}

func NewTerrain(options ...func(*Config)) *Terrain {
        var t Terrain
        for _, option := range options {
                option(&t.config)
        }
        return &t
}
```

01:30

To begin, I'll very quickly recap the functional options pattern

[click] We start with some options, expressed as functions which take a pointer to a structure to configure.

[click] We pass those functions to a constructor, and inside the body of that constructor each option function is invoked in order, passing in a reference to the Config value.

[click] finally we call NewTerrain with the options we want. And away we go

Ok, everyone should be familiar with this pattern.

Where I believe the confusion comes from, is when you need an option function which take a parameter.

```
func WithReticulatedSplines(c *Config) { ... }

type Terrain struct {
        config Config
}

func NewTerrain(options ...func(*Config)) *Terrain {
        var t Terrain
        for _, option := range options {
                option(&t.config)
        }
        return &t
}

func main() {
        t := NewTerrain(WithReticulatedSplines)
        // [ simulation intensifies ]
}
```

01:30

To begin, I'll very quickly recap the functional options pattern

[click] We start with some options, expressed as functions which take a pointer to a structure to configure.

[click] We pass those functions to a constructor, and inside the body of that constructor each option function is invoked in order, passing in a reference to the Config value.

[click] finally we call NewTerrain with the options we want. And away we go

Ok, everyone should be familiar with this pattern.

Where I believe the confusion comes from, is when you need an option function which take a parameter.

```
// WithCities adds n cities to the Terrain model
func WithCities(n int) func(*Config) { ... }
```

02:00

For example, we have WithCities, which lets us add a number of cities to our terrain model.

Because WithCities takes an argument, we cannot simply pass WithCities to NewTerrain — the signature does not match.

[click] Instead we evaluate WithCities, passing in the number of cities to create, and use the result of this function as the value to pass to NewTerrain.

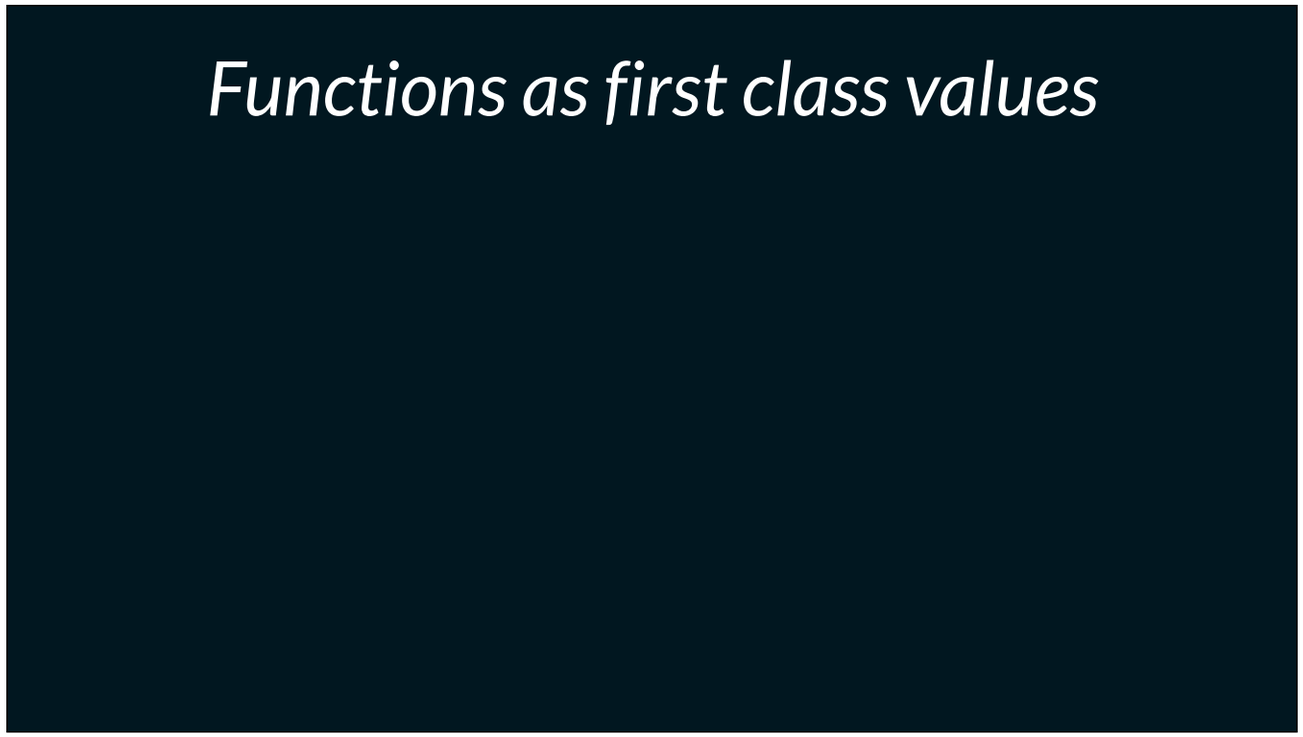
```
// WithCities adds n cities to the Terrain model
func WithCities(n int) func(*Config) { ... }

func main() {
        t := NewTerrain(WithCities(9))
        // ...
}
```

02:00

For example, we have WithCities, which lets us add a number of cities to our terrain model.

Because WithCities takes an argument, we cannot simply pass WithCities to NewTerrain — the signature does not match.

[click] Instead we evaluate WithCities, passing in the number of cities to create, and use the result of this function as the value to pass to NewTerrain.



02:30

What’s going on here? Let’s break it down.

fundamentally: evaluating a function returns a value.

[click] We have functions that take two numbers and return a number

[click] We have functions that take a slice, and return a pointer to a structure.

[click] and now we have a function which returns a function.

The type of the value that is returned from WithCities is a function which takes a pointer to a Config. This ability to treat functions as regular values leads to their name, first class functions.

Functions as first class values

```
package math

func Min(a, b float64) float64
```

02:30

What’s going on here? Let’s break it down.

fundamentally: evaluating a function returns a value.

[click] We have functions that take two numbers and return a number

[click] We have functions that take a slice, and return a pointer to a structure.

[click] and now we have a function which returns a function.

The type of the value that is returned from WithCities is a function which takes a pointer to a Config. This ability to treat functions as regular values leads to their name, first class functions.

Functions as first class values

```
package math

func Min(a, b float64) float64
```

```
package bytes

func NewReader(b []byte) *Reader
```

02:30

What’s going on here? Let’s break it down.

fundamentally: evaluating a function returns a value.

[click] We have functions that take two numbers and return a number

[click] We have functions that take a slice, and return a pointer to a structure.

[click] and now we have a function which returns a function.

The type of the value that is returned from WithCities is a function which takes a pointer to a Config. This ability to treat functions as regular values leads to their name, first class functions.

Functions as first class values

```
package math

func Min(a, b float64) float64

package bytes

func NewReader(b []byte) *Reader

func WithCities(n int) func(*Config)
```

02:30

What’s going on here? Let’s break it down.

fundamentally: evaluating a function returns a value.

[click] We have functions that take two numbers and return a number

[click] We have functions that take a slice, and return a pointer to a structure.

[click] and now we have a function which returns a function.

The type of the value that is returned from WithCities is a function which takes a pointer to a Config. This ability to treat functions as regular values leads to their name, first class functions.

interface.Apply

Another way to think about what is going on here is to try to rewrite the functional option pattern using an interface.

[click] Rather than using a function type, we declare an interface, we'll call it Option and give it a single method, Apply which takes a pointer to Config.

[click]Whenever we call NewTerrain we pass in one or more values that implement the Option interface. Inside NewTerrain, just as before, we loop over the slice of options and call the Apply method on each.

Ok, so this doesn't look too different to the previous example, rather than ranging over a slice of functions and calling them, we range over a slice of interface values and call a method on each.

Let's take a look at the other side, declaring an option.

interface.Apply

```
type Option interface {
        Apply(*Config)
}
```

Another way to think about what is going on here is to try to rewrite the functional option pattern using an interface.

[click] Rather than using a function type, we declare an interface, we'll call it Option and give it a single method, Apply which takes a pointer to Config.

[click]Whenever we call NewTerrain we pass in one or more values that implement the Option interface. Inside NewTerrain, just as before, we loop over the slice of options and call the Apply method on each.

Ok, so this doesn't look too different to the previous example, rather than ranging over a slice of functions and calling them, we range over a slice of interface values and call a method on each.

Let's take a look at the other side, declaring an option.

interface.Apply

```
type Option interface {
        Apply(*Config)
}

func NewTerrain(options ...Option) *Terrain {
        var config Config
        for _, option := range options {
                option.Apply(&config)
        }
        // ...
}
```

Another way to think about what is going on here is to try to rewrite the functional option pattern using an interface.

[click] Rather than using a function type, we declare an interface, we'll call it Option and give it a single method, Apply which takes a pointer to Config.

[click]Whenever we call NewTerrain we pass in one or more values that implement the Option interface. Inside NewTerrain, just as before, we loop over the slice of options and call the Apply method on each.

Ok, so this doesn't look too different to the previous example, rather than ranging over a slice of functions and calling them, we range over a slice of interface values and call a method on each.

Let's take a look at the other side, declaring an option.

```
type splines struct {}

func (s *splines) Apply(c *Config) { ... }
```

04:00

Because we’re passing around interface implementations, we need to declare a type to hold the Apply method.

[click] We also need to declare a constructor function to return our splines option implementation.

You can already see that this is going to be more code.

```
type splines struct {}

func (s *splines) Apply(c *Config) { ... }

func WithReticulatedSplines() Option {
        return new(splines)
}
```

04:00

Because we’re passing around interface implementations, we need to declare a type to hold the Apply method.

[click] We also need to declare a constructor function to return our splines option implementation.

You can already see that this is going to be more code.

```
type cities struct {
        cities int
}

func (c *cities) Apply(c *Config) { ... }
```

Now to write WithCities using our Option interface we need to do a bit more work.

In the previous functional version, the value of n; the number of cities to create, was captured lexically for us in the declaration of the anonymous function.

[click] because we're using an interface we need to declare a type to hold the count of cities and we need a constructor to assign the field during construction.

[click] Finally, putting it all together, we call NewTerrain with the results of evaluating WithReticulatedSplines and WithCities.

05:00

At GopherCon last year Tomas Senart spoke about this duality of a first class function and an interface with one method. You can see this play out in the example; an interface with one method, and a function are equivalent.

But, as you can see, dealing with functions as a first class citizen involves much less code.

```
type cities struct {
        cities int
}

func (c *cities) Apply(c *Config) { ... }

func WithCities(n int) Option {
        return &cities{
                cities: n,
        }
}
```

Now to write WithCities using our Option interface we need to do a bit more work.

In the previous functional version, the value of n; the number of cities to create, was captured lexically for us in the declaration of the anonymous function.

[click] because we're using an interface we need to declare a type to hold the count of cities and we need a constructor to assign the field during construction.

[click] Finally, putting it all together, we call NewTerrain with the results of evaluating WithReticulatedSplines and WithCities.

05:00

At GopherCon last year Tomas Senart spoke about this duality of a first class function and an interface with one method. You can see this play out in the example; an interface with one method, and a function are equivalent.

But, as you can see, dealing with functions as a first class citizen involves much less code.

```
func (c *cities) Apply(c *Config) { ... }

func WithCities(n int) Option {
        return &cities{
                cities: n,
        }
}

func main() {
        t := NewTerrain(
                WithReticulatedSplines(),
                WithCities(9),
        )
        // ...
}
```

Now to write WithCities using our Option interface we need to do a bit more work.

In the previous functional version, the value of n; the number of cities to create, was captured lexically for us in the declaration of the anonymous function.

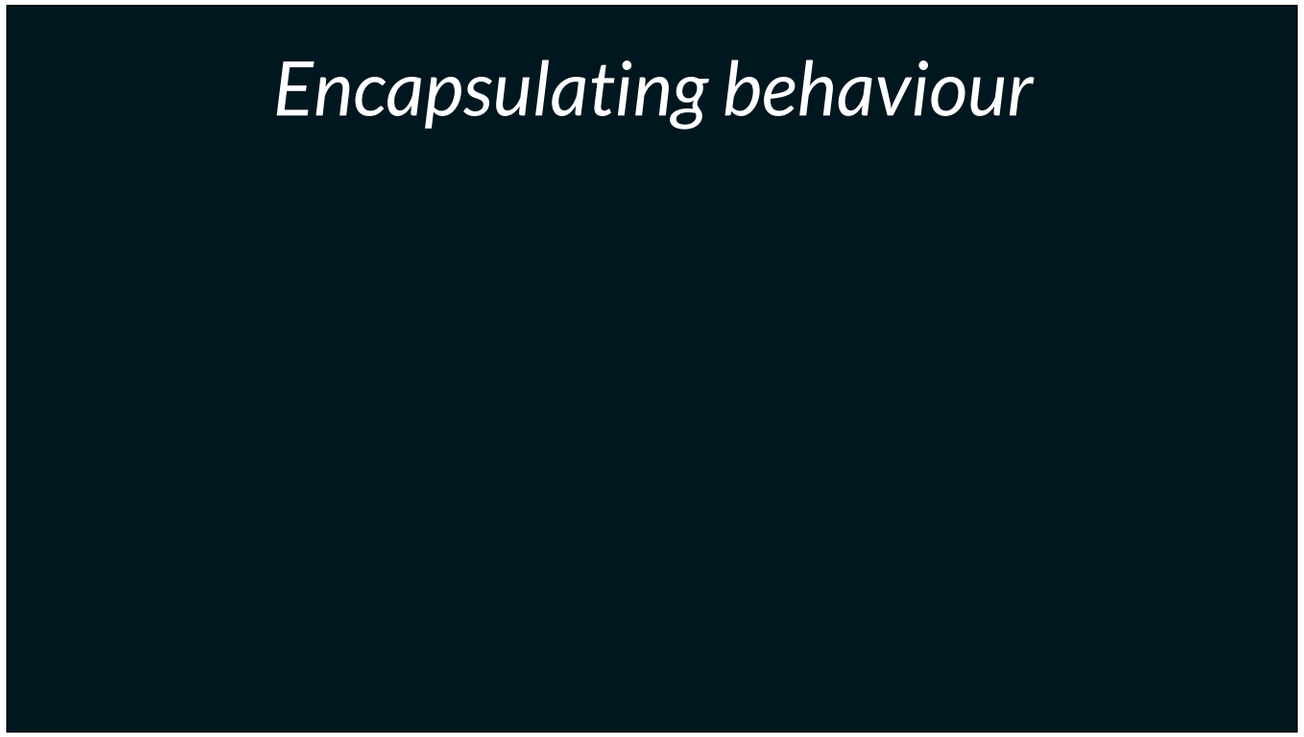
[click] because we're using an interface we need to declare a type to hold the count of cities and we need a constructor to assign the field during construction.

[click] Finally, putting it all together, we call NewTerrain with the results of evaluating WithReticulatedSplines and WithCities.

05:00

At GopherCon last year Tomas Senart spoke about this duality of a first class function and an interface with one method. You can see this play out in the example; an interface with one method, and a function are equivalent.

But, as you can see, dealing with functions as a first class citizen involves much less code.



05:30

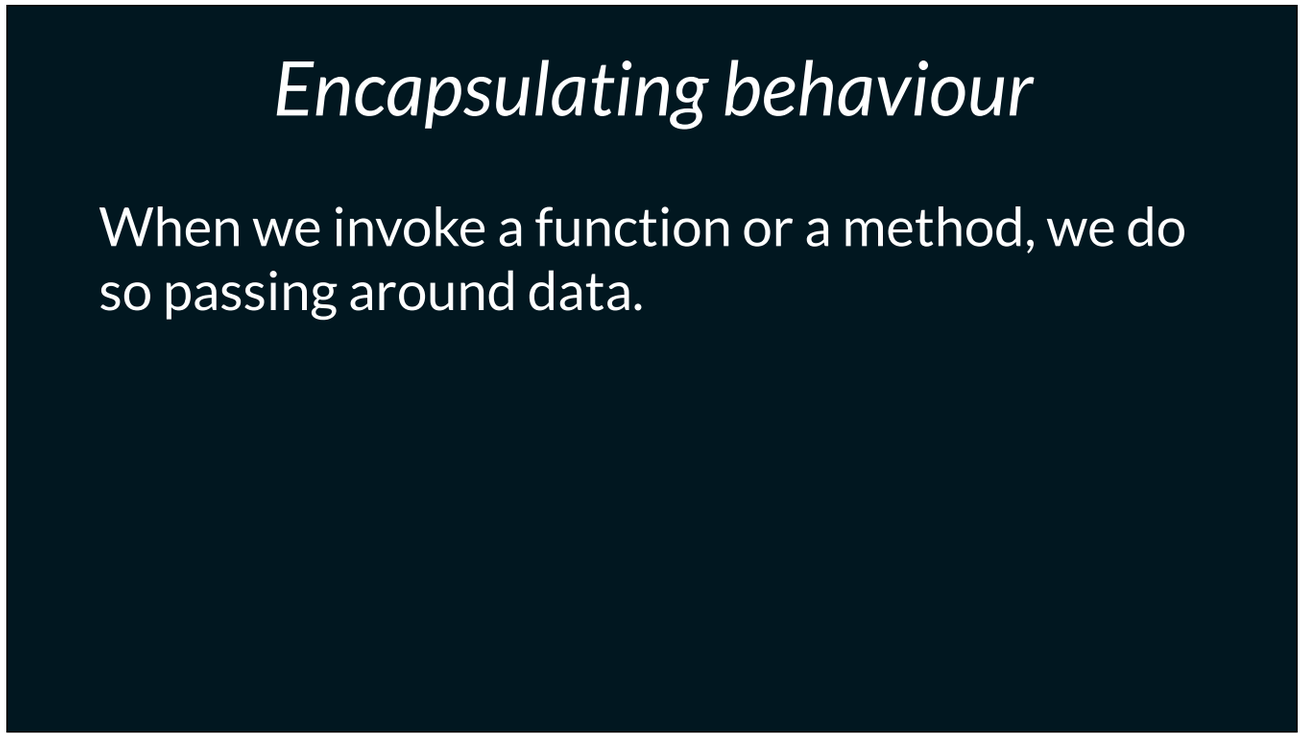
Ok, let’s leave interfaces for a moment and talk about some other properties of first class functions.

[click] when we invoke a function or a method, we do so passing around data. The job of that function is often to interpret that data and take some action.

[click] Function values allow you to pass behaviour to be executed, rather that data to be interpreted.

[click] In effect, passing a function value allows you to declare code that will execute later, perhaps in a different context.

To illustrate this, let’s look at some examples.



05:30

Ok, let’s leave interfaces for a moment and talk about some other properties of first class functions.

[click] when we invoke a function or a method, we do so passing around data. The job of that function is often to interpret that data and take some action.

[click] Function values allow you to pass behaviour to be executed, rather that data to be interpreted.

[click] In effect, passing a function value allows you to declare code that will execute later, perhaps in a different context.

To illustrate this, let’s look at some examples.

Encapsulating behaviour

When we invoke a function or a method, we do so passing around data.

Function values allow you to pass behaviour to be executed, rather that data to be interpreted.

05:30

Ok, let’s leave interfaces for a moment and talk about some other properties of first class functions.

[click] when we invoke a function or a method, we do so passing around data. The job of that function is often to interpret that data and take some action.

[click] Function values allow you to pass behaviour to be executed, rather that data to be interpreted.

[click] In effect, passing a function value allows you to declare code that will execute later, perhaps in a different context.

To illustrate this, let’s look at some examples.

Encapsulating behaviour

When we invoke a function or a method, we do so passing around data.

Function values allow you to pass behaviour to be executed, rather that data to be interpreted.

In effect, function values allows you to declare code that will execute in a different context.

05:30

Ok, let’s leave interfaces for a moment and talk about some other properties of first class functions.

[click] when we invoke a function or a method, we do so passing around data. The job of that function is often to interpret that data and take some action.

[click] Function values allow you to pass behaviour to be executed, rather that data to be interpreted.

[click] In effect, passing a function value allows you to declare code that will execute later, perhaps in a different context.

To illustrate this, let’s look at some examples.

```
type Calculator struct {
        acc float64
}
```

06:30

Here is a simple calculator type.

[click] It has a set of operations it understands

[click] It has one method, Do, which takes an operation and an operand, v.

For convenience, Do also returns the value of the accumulator after the operation is applied.

[click] Our calculator only knows how to add, subtract, and multiply.

If we wanted to implement division, we'd have to allocate an operation constant, then open up the Do method and add the code to implement division.

Sounds reasonable, it's only a few lines, but what if we wanted to add square root, or exponentiation?

Each time we do this, Do is going to grow longer, and become harder to follow, Because each time we add an operation, we have to encode into the Do method knowledge of how to *interpret* that operation.

Let's rewrite our calculator a little

```
type Calculator struct {
        acc float64
}

const (
        OP_ADD = 1 << iota
        OP_SUB
        OP_MUL
)
```

06:30

Here is a simple calculator type.

[click] It has a set of operations it understands

[click] It has one method, Do, which takes an operation and an operand, v.

For convenience, Do also returns the value of the accumulator after the operation is applied.

[click] Our calculator only knows how to add, subtract, and multiply.

If we wanted to implement division, we'd have to allocate an operation constant, then open up the Do method and add the code to implement division.

Sounds reasonable, it's only a few lines, but what if we wanted to add square root, or exponentiation?

Each time we do this, Do is going to grow longer, and become harder to follow, Because each time we add an operation, we have to encode into the Do method knowledge of how to *interpret* that operation.

Let's rewrite our calculator a little

```
const (
        OP_ADD = 1 << iota
        OP_SUB
        OP_MUL
)

func (c *Calculator) Do(op int, v float64) float64 {
        switch op {
        case OP_ADD:
                c.acc += v
        case OP_SUB:
                c.acc -= v
        case OP_MUL:
                c.acc *= v
        default:
                panic("unhandled operation")
        }
        return c.acc
}
```

06:30

Here is a simple calculator type.

[click] It has a set of operations it understands

[click] It has one method, Do, which takes an operation and an operand, v.

For convenience, Do also returns the value of the accumulator after the operation is applied.

[click] Our calculator only knows how to add, subtract, and multiply.

If we wanted to implement division, we'd have to allocate an operation constant, then open up the Do method and add the code to implement division.

Sounds reasonable, it's only a few lines, but what if we wanted to add square root, or exponentiation?

Each time we do this, Do is going to grow longer, and become harder to follow, Because each time we add an operation, we have to encode into the Do method knowledge of how to *interpret* that operation.

Let's rewrite our calculator a little

```
func (c *Calculator) Do(op int, v float64) float64 {
        switch op {
        case OP_ADD:
                c.acc += v
        case OP_SUB:
                c.acc -= v
        case OP_MUL:
                c.acc *= v
        default:
                panic("unhandled operation")
        }
        return c.acc
}

func main() {
        var c Calculator
        fmt.Println(c.Do(OP_ADD, 100))     // 100
        fmt.Println(c.Do(OP_SUB, 50))      // 50
        fmt.Println(c.Do(OP_MUL, 2))       // 100
}
```

06:30

Here is a simple calculator type.

[click] It has a set of operations it understands

[click] It has one method, Do, which takes an operation and an operand, v.

For convenience, Do also returns the value of the accumulator after the operation is applied.

[click] Our calculator only knows how to add, subtract, and multiply.

If we wanted to implement division, we'd have to allocate an operation constant, then open up the Do method and add the code to implement division.

Sounds reasonable, it's only a few lines, but what if we wanted to add square root, or exponentiation?

Each time we do this, Do is going to grow longer, and become harder to follow, Because each time we add an operation, we have to encode into the Do method knowledge of how to *interpret* that operation.

Let's rewrite our calculator a little

```
type Calculator struct
        acc float64
}
```

07:30

As before we have a Calculator, which manages its own accumulator.

[click] The Calculator has a Do method, which this time takes an function as the operation, and a value as the operand.

This signature the operation function could be intimidating, so i’ve broken it out into it’s own declaration.

let’s talk about that quickly

The opfunc type is a function which takes two float64’s and returns a third.

Whenever Do is called, it calls the operation we pass in, using its own accumulator and the operand we provide.

So, how do we use this new Calculator?

[click] You guessed it, by writing our operations as functions.

This is the code for Add, what about the other operations? It turns out they aren’t too hard either.

```
type Calculator struct
        acc float64
}

type opfunc func(float64, float64) float64

func (c *Calculator) Do(op opfunc, v float64) float64 {
        c.acc = op(c.acc, v)
        return c.acc
}
```

07:30

As before we have a Calculator, which manages its own accumulator.

[click] The Calculator has a Do method, which this time takes an function as the operation, and a value as the operand.

This signature the operation function could be intimidating, so i’ve broken it out into it’s own declaration.

let’s talk about that quickly

The opfunc type is a function which takes two float64’s and returns a third.

Whenever Do is called, it calls the operation we pass in, using its own accumulator and the operand we provide.

So, how do we use this new Calculator?

[click] You guessed it, by writing our operations as functions.

This is the code for Add, what about the other operations? It turns out they aren’t too hard either.

```
type Calculator struct
        acc float64
}

type opfunc func(float64, float64) float64

func (c *Calculator) Do(op opfunc, v float64) float64 {
        c.acc = op(c.acc, v)
        return c.acc
}

func Add(a, b float64) float64 {
        return a + b
}
```

07:30

As before we have a Calculator, which manages its own accumulator.

[click] The Calculator has a Do method, which this time takes an function as the operation, and a value as the operand.

This signature the operation function could be intimidating, so i’ve broken it out into it’s own declaration.

let’s talk about that quickly

The opfunc type is a function which takes two float64’s and returns a third.

Whenever Do is called, it calls the operation we pass in, using its own accumulator and the operand we provide.

So, how do we use this new Calculator?

[click] You guessed it, by writing our operations as functions.

This is the code for Add, what about the other operations? It turns out they aren’t too hard either.

```
type Calculator struct
        acc float64
}

type opfunc func(float64, float64) float64

func (c *Calculator) Do(op opfunc, v float64) float64 {
        c.acc = op(c.acc, v)
        return c.acc
}

func Add(a, b float64) float64 {
        return a + b
}

func Sub(a, b float64) float64 {
        return a - b
}
```

07:30

As before we have a Calculator, which manages its own accumulator.

[click] The Calculator has a Do method, which this time takes an function as the operation, and a value as the operand.

This signature the operation function could be intimidating, so i’ve broken it out into it’s own declaration.

let’s talk about that quickly

The opfunc type is a function which takes two float64’s and returns a third.

Whenever Do is called, it calls the operation we pass in, using its own accumulator and the operand we provide.

So, how do we use this new Calculator?

[click] You guessed it, by writing our operations as functions.

This is the code for Add, what about the other operations? It turns out they aren’t too hard either.

```
type opfunc func(float64, float64) float64

func (c *Calculator) Do(op opfunc, v float64) float64 {
        c.acc = op(c.acc, v)
        return c.acc
}

func Add(a, b float64) float64 {
        return a + b
}

func Sub(a, b float64) float64 {
        return a - b
}

func Mul(a, b float64) float64 {
        return a * b
}
```

07:30

As before we have a Calculator, which manages its own accumulator.

[click] The Calculator has a Do method, which this time takes an function as the operation, and a value as the operand.

This signature the operation function could be intimidating, so i’ve broken it out into it’s own declaration.

let’s talk about that quickly

The opfunc type is a function which takes two float64’s and returns a third.

Whenever Do is called, it calls the operation we pass in, using its own accumulator and the operand we provide.

So, how do we use this new Calculator?

[click] You guessed it, by writing our operations as functions.

This is the code for Add, what about the other operations? It turns out they aren’t too hard either.

```
func Add(a, b float64) float64 {
        return a + b
}

func Sub(a, b float64) float64 {
        return a - b
}

func Mul(a, b float64) float64 {
        return a * b
}

func main() {
        var c Calculator
        fmt.Println(c.Do(Add, 5))       // 5
        fmt.Println(c.Do(Sub, 3))       // 2
        fmt.Println(c.Do(Mul, 8))       // 16
}
```

07:30

As before we have a Calculator, which manages its own accumulator.

[click] The Calculator has a Do method, which this time takes an function as the operation, and a value as the operand.

This signature the operation function could be intimidating, so i’ve broken it out into it’s own declaration.

let’s talk about that quickly

The opfunc type is a function which takes two float64’s and returns a third.

Whenever Do is called, it calls the operation we pass in, using its own accumulator and the operand we provide.

So, how do we use this new Calculator?

[click] You guessed it, by writing our operations as functions.

This is the code for Add, what about the other operations? It turns out they aren’t too hard either.

```
func Sqrt(n, _ float64) float64 {
        return math.Sqrt(n)
}
```

09:00

But, it turns out there is a problem. Square root take one argument, not two. But our calculator’s Do method signature requires an operation function that takes two arguments.

[click] Maybe we just cheat and ignore the operand. That’s a bit gross, I think we can do better.

```
func Sqrt(n, _ float64) float64 {
        return math.Sqrt(n)
}

func main() {
        var c Calculator
        c.Do(Add, 16)
        c.Do(Sqrt, 0) // operand ignored
}
```

09:00

But, it turns out there is a problem. Square root take one argument, not two. But our calculator’s Do method signature requires an operation function that takes two arguments.

[click] Maybe we just cheat and ignore the operand. That’s a bit gross, I think we can do better.

```
func Add(n float64) func(float64) float64 {
        return func(acc float64) float64 {
                return acc + n
        }
}
```

09:30

Let’s redefine Add from a function that is called with two values and returns a third, to a function which *returns* a function that takes a value and returns a value.

[click] calc.Do now invokes the operation function passing in its own accumulator and recording the result back in the accumulator.

[click] Now in main we call calc.Do with not with the Add function itself, but with the result of evaluating Add(10).

The type of evaluating Add(10) is a function which takes a value, and returns a value, matching the signature that Do requires.

```
func Add(n float64) func(float64) float64 {
        return func(acc float64) float64 {
                return acc + n
        }
}

func (c *Calculator) Do(op func(float64) float64) float64 {
        c.acc = op(c.acc)
        return c.acc
}
```

09:30

Let’s redefine Add from a function that is called with two values and returns a third, to a function which *returns* a function that takes a value and returns a value.

[click] calc.Do now invokes the operation function passing in its own accumulator and recording the result back in the accumulator.

[click] Now in main we call calc.Do with not with the Add function itself, but with the result of evaluating Add(10).

The type of evaluating Add(10) is a function which takes a value, and returns a value, matching the signature that Do requires.

```
func Add(n float64) func(float64) float64 {
        return func(acc float64) float64 {
                return acc + n
        }
}

func (c *Calculator) Do(op func(float64) float64) float64 {
        c.acc = op(c.acc)
        return c.acc
}

func main() {
        var c Calculator
        c.Do(Add(10))   // 10
        c.Do(Add(20))   // 30
}
```

09:30

Let’s redefine Add from a function that is called with two values and returns a third, to a function which *returns* a function that takes a value and returns a value.

[click] calc.Do now invokes the operation function passing in its own accumulator and recording the result back in the accumulator.

[click] Now in main we call calc.Do with not with the Add function itself, but with the result of evaluating Add(10).

The type of evaluating Add(10) is a function which takes a value, and returns a value, matching the signature that Do requires.

```
func Sub(n float64) func(float64) float64 {
        return func(acc float64) float64 {
                return acc - n
        }
}

func Mul(n float64) func(float64) float64 {
        return func(acc float64) float64 {
                return acc * n
        }
}
```

10:00

Subtraction and multiplication are similarly easy to implement.

But what about square root?

[click] Now this is easy

This implementation of square root avoids the awkward syntax of the previous calculator's operation function, as our calculator now operates on functions which take and return one value.

[click]

Hopefully you've noticed that the signature of our Sqrt function is the same as math.Sqrt, so we can make this code smaller by reusing any function from the math package that takes a single argument.

```
func Sub(n float64) func(float64) float64 {
        return func(acc float64) float64 {
                return acc - n
        }
}

func Mul(n float64) func(float64) float64 {
        return func(acc float64) float64 {
                return acc * n
        }
}

func Sqrt() func(float64) float64 {
        return func(n float64) float64 {
                return math.Sqrt(n)
        }
}
```

10:00

Subtraction and multiplication are similarly easy to implement.

But what about square root?

[click] Now this is easy

This implementation of square root avoids the awkward syntax of the previous calculator's operation function, as our calculator now operates on functions which take and return one value.

[click]

Hopefully you've noticed that the signature of our Sqrt function is the same as math.Sqrt, so we can make this code smaller by reusing any function from the math package that takes a single argument.

```
func Mul(n float64) func(float64) float64 {
        return func(acc float64) float64 {
                return acc * n
        }
}

func Sqrt() func(float64) float64 {
        return func(n float64) float64 {
                return math.Sqrt(n)
        }
}

func main() {
        var c Calculator
        c.Do(Add(2))
        c.Do(Sqrt())   // 1.41421356237
}
```

10:00

Subtraction and multiplication are similarly easy to implement.

But what about square root?

[click] Now this is easy

This implementation of square root avoids the awkward syntax of the previous calculator's operation function, as our calculator now operates on functions which take and return one value.

[click]

Hopefully you've noticed that the signature of our Sqrt function is the same as math.Sqrt, so we can make this code smaller by reusing any function from the math package that takes a single argument.

```
func main() {
        var c Calculator
        c.Do(Add(2))      // 2
        c.Do(math.Sqrt)   // 1.41421356237
        c.Do(math.Cos)    // 0.99969539804
}
```

We started with a model of hard coded, interpreted logic.

We moved to a functional model, where we pass in the behaviour we want.

Then, by taking it a step further, we generalised our calculator to work for operations regardless of their number of arguments.

11:00

Let’s change tracks a little and talk about why most of us are here at a Go conference; concurrency, specifically actors.

And to give due credit, the examples here are inspired by Bryan Boreham’s talk from GolangUK, you should check it out.

Suppose we’re building a chat server, we plan to be the next Hipchat or Slack, but we’ll start small for the moment.

```
type Mux struct {
        mu    sync.Mutex
        conns map[net.Addr]net.Conn
}
```

11:30

Here’s the first cut of the heart of any chat system.

[click] We have a way to register new connections

[click] remove old ones, and

[click] send a message to all the registered connections.

Now, because this is a server, all of these methods will be called concurrently so we need to use a Mutex to protect the conns map and prevent data races.

Is this what you’d call idiomatic Go code? Maybe?

```
type Mux struct {
        mu    sync.Mutex
        conns map[net.Addr]net.Conn
}

func (m *Mux) Add(conn net.Conn) {
        m.mu.Lock()
        defer m.mu.Unlock()
        m.conns[conn.RemoteAddr()] = conn
}
```

11:30

Here’s the first cut of the heart of any chat system.

[click] We have a way to register new connections

[click] remove old ones, and

[click] send a message to all the registered connections.

Now, because this is a server, all of these methods will be called concurrently so we need to use a Mutex to protect the conns map and prevent data races.

Is this what you’d call idiomatic Go code? Maybe?

```
type Mux struct {
        mu    sync.Mutex
        conns map[net.Addr]net.Conn
}

func (m *Mux) Add(conn net.Conn) {
        m.mu.Lock()
        defer m.mu.Unlock()
        m.conns[conn.RemoteAddr()] = conn
}

func (m *Mux) Remove(addr net.Addr) {
        m.mu.Lock()
        defer m.mu.Unlock()
        delete(m.conns, addr)
}
```

11:30

Here’s the first cut of the heart of any chat system.

[click] We have a way to register new connections

[click] remove old ones, and

[click] send a message to all the registered connections.

Now, because this is a server, all of these methods will be called concurrently so we need to use a Mutex to protect the conns map and prevent data races.

Is this what you’d call idiomatic Go code? Maybe?

```
func (m *Mux) Remove(addr net.Addr) {
        m.mu.Lock()
        defer m.mu.Unlock()
        delete(m.conns, addr)
}

func (m *Mux) SendMsg(msg string) error {
        m.mu.Lock()
        defer m.mu.Unlock()
        for _, conn := range m.conns {
                err := io.WriteString(conn, msg)
                if err != nil {
                        return err
                }
        }
        return nil
}
```

11:30

Here’s the first cut of the heart of any chat system.

[click] We have a way to register new connections

[click] remove old ones, and

[click] send a message to all the registered connections.

Now, because this is a server, all of these methods will be called concurrently so we need to use a Mutex to protect the conns map and prevent data races.

Is this what you’d call idiomatic Go code? Maybe?

12:00

Our first proverb.

Don’t mediate access to shared memory with locks and mutexes, instead share that memory by communicating.

So let’s apply this advice to our chat server.

```
type Mux struct {
        add     chan net.Conn
        remove  chan net.Addr
        sendMsg chan string
}
```

Rather than using a mutex to serialise access to the Mux's conns map, we can give that job to a goroutine, and communicate with that goroutine via channels.

[click] Add sends the connection to add to the add channel

[click] Remove sends the address of the connection to the remove channel

[click] And send message sends the message to be transmitted to each connection to the sendMsg channel.

[click] Rather than using a mutex to serialise access to the conns map, loop will wait until it receives an operation, in the form of a value sent over one of the add, remove, or sendMsg channels and apply the relevant case.

We don’t need a mutex any more, because the shared state, our conns map, is local to the loop function.

But, there’s still a lot of hard coded logic here. loop only knows how to do three things; add, remove and broadcast a message. As with the previous calculator example, adding new features to our Mux type will involve creating

- a channel
- adding a helper to send the data over the channel
- and extending the select logic inside loop to process that data.

```
type Mux struct {
        add     chan net.Conn
        remove  chan net.Addr
        sendMsg chan string
}

func (m *Mux) Add(conn net.Conn) {
        m.add <- conn
}
```

Rather than using a mutex to serialise access to the Mux's conns map, we can give that job to a goroutine, and communicate with that goroutine via channels.

[click] Add sends the connection to add to the add channel

[click] Remove sends the address of the connection to the remove channel

[click] And send message sends the message to be transmitted to each connection to the sendMsg channel.

[click] Rather than using a mutex to serialise access to the conns map, loop will wait until it receives an operation, in the form of a value sent over one of the add, remove, or sendMsg channels and apply the relevant case.

We don’t need a mutex any more, because the shared state, our conns map, is local to the loop function.

But, there’s still a lot of hard coded logic here. loop only knows how to do three things; add, remove and broadcast a message. As with the previous calculator example, adding new features to our Mux type will involve creating

- a channel
- adding a helper to send the data over the channel
- and extending the select logic inside loop to process that data.

```
type Mux struct {
        add     chan net.Conn
        remove  chan net.Addr
        sendMsg chan string
}

func (m *Mux) Add(conn net.Conn) {
        m.add <- conn
}

func (m *Mux) Remove(addr net.Addr) {
        m.remove <- add
}
```

Rather than using a mutex to serialise access to the Mux's conns map, we can give that job to a goroutine, and communicate with that goroutine via channels.

[click] Add sends the connection to add to the add channel

[click] Remove sends the address of the connection to the remove channel

[click] And send message sends the message to be transmitted to each connection to the sendMsg channel.

[click] Rather than using a mutex to serialise access to the conns map, loop will wait until it receives an operation, in the form of a value sent over one of the add, remove, or sendMsg channels and apply the relevant case.

We don’t need a mutex any more, because the shared state, our conns map, is local to the loop function.

But, there’s still a lot of hard coded logic here. loop only knows how to do three things; add, remove and broadcast a message. As with the previous calculator example, adding new features to our Mux type will involve creating

- a channel
- adding a helper to send the data over the channel
- and extending the select logic inside loop to process that data.

```
type Mux struct {
        add     chan net.Conn
        remove  chan net.Addr
        sendMsg chan string
}

func (m *Mux) Add(conn net.Conn) {
        m.add <- conn
}

func (m *Mux) Remove(addr net.Addr) {
        m.remove <- add
}

func (m *Mux) SendMsg(msg string) error {
        m.sendMsg <- msg
        return nil
}
```

Rather than using a mutex to serialise access to the Mux's conns map, we can give that job to a goroutine, and communicate with that goroutine via channels.

[click] Add sends the connection to add to the add channel

[click] Remove sends the address of the connection to the remove channel

[click] And send message sends the message to be transmitted to each connection to the sendMsg channel.

[click] Rather than using a mutex to serialise access to the conns map, loop will wait until it receives an operation, in the form of a value sent over one of the add, remove, or sendMsg channels and apply the relevant case.

We don’t need a mutex any more, because the shared state, our conns map, is local to the loop function.

But, there’s still a lot of hard coded logic here. loop only knows how to do three things; add, remove and broadcast a message. As with the previous calculator example, adding new features to our Mux type will involve creating

- a channel
- adding a helper to send the data over the channel
- and extending the select logic inside loop to process that data.

```
        m.sendMsg <- msg
        return nil
}

func (m *Mux) loop() {
        conns := make(map[net.Addr]net.Conn)
        for {
                select {
                case conn := <-m.add:
                        m.conns[conn.RemoteAddr()] = conn
                case addr := <-m.remove:
                        delete(m.conns, addr)
                case msg := <-m.sendMsg:
                        for _, conn := range m.conns {
                                io.WriteString(conn, msg)
                        }
                }
        }
}
```

Rather than using a mutex to serialise access to the Mux's conns map, we can give that job to a goroutine, and communicate with that goroutine via channels.

[click] Add sends the connection to add to the add channel

[click] Remove sends the address of the connection to the remove channel

[click] And send message sends the message to be transmitted to each connection to the sendMsg channel.

[click] Rather than using a mutex to serialise access to the conns map, loop will wait until it receives an operation, in the form of a value sent over one of the add, remove, or sendMsg channels and apply the relevant case.

We don’t need a mutex any more, because the shared state, our conns map, is local to the loop function.

But, there’s still a lot of hard coded logic here. loop only knows how to do three things; add, remove and broadcast a message. As with the previous calculator example, adding new features to our Mux type will involve creating

- a channel
- adding a helper to send the data over the channel
- and extending the select logic inside loop to process that data.

```
type Mux struct {
        ops chan func(map[net.Addr]net.Conn)
}
```

Now, each method sends an operation to be executed in the context of the loop function, using our single ops channel.

[click] In this case the signature of the operation is a function which takes a map of net.Addr's to net.Conn's. In a real program you'd probably have a much more complicated type to represent a client connection, but it's sufficient for the purpose of this example.

[click] remove is similar, we send a function that deletes it's connection's address from the supplied map

[click] send message is a function which iterates over all connections in the supplied map and calls write string to send each a copy of the message.

[click] You can see that we've moved the logic from the body of loop into anonymous functions created by our helpers.

So the job of loop is now to create a conns map, then waits for an operation to be provided on the ops channel, and invokes it, passing in the conns map.

But there are a few problems still to fix. The most pressing is the lack of error handling in SendMsg; an error writing to a connection will not be communicated back to the caller. So let's fix that now

```
type Mux struct {
        ops chan func(map[net.Addr]net.Conn)
}

func (m *Mux) Add(conn net.Conn) {
        m.ops <- func(m map[net.Addr]net.Conn) {
                m[conn.RemoteAddr()] = conn
        }
}
```

Now, each method sends an operation to be executed in the context of the loop function, using our single ops channel.

[click] In this case the signature of the operation is a function which takes a map of net.Addr's to net.Conn's. In a real program you'd probably have a much more complicated type to represent a client connection, but it's sufficient for the purpose of this example.

[click] remove is similar, we send a function that deletes it's connection's address from the supplied map

[click] send message is a function which iterates over all connections in the supplied map and calls write string to send each a copy of the message.

[click] You can see that we've moved the logic from the body of loop into anonymous functions created by our helpers.

So the job of loop is now to create a conns map, then waits for an operation to be provided on the ops channel, and invokes it, passing in the conns map.

But there are a few problems still to fix. The most pressing is the lack of error handling in SendMsg; an error writing to a connection will not be communicated back to the caller. So let's fix that now

```
type Mux struct {
        ops chan func(map[net.Addr]net.Conn)
}

func (m *Mux) Add(conn net.Conn) {
        m.ops <- func(m map[net.Addr]net.Conn) {
                m[conn.RemoteAddr()] = conn
        }
}

func (m *Mux) Remove(addr net.Addr) {
        m.ops <- func(m map[net.Addr]net.Conn) {
                delete(m, addr)
        }
}
```

Now, each method sends an operation to be executed in the context of the loop function, using our single ops channel.

[click] In this case the signature of the operation is a function which takes a map of net.Addr's to net.Conn's. In a real program you'd probably have a much more complicated type to represent a client connection, but it's sufficient for the purpose of this example.

[click] remove is similar, we send a function that deletes it's connection's address from the supplied map

[click] send message is a function which iterates over all connections in the supplied map and calls write string to send each a copy of the message.

[click] You can see that we've moved the logic from the body of loop into anonymous functions created by our helpers.

So the job of loop is now to create a conns map, then waits for an operation to be provided on the ops channel, and invokes it, passing in the conns map.

But there are a few problems still to fix. The most pressing is the lack of error handling in SendMsg; an error writing to a connection will not be communicated back to the caller. So let's fix that now

```
}

func (m *Mux) Remove(addr net.Addr) {
        m.ops <- func(m map[net.Addr]net.Conn) {
                delete(m, addr)
        }
}

func (m *Mux) SendMsg(msg string) error {
        m.ops <- func(m map[net.Addr]net.Conn) {
                for _, conn := range m {
                        io.WriteString(conn, msg)
                }
        }
        return nil
}
```

Now, each method sends an operation to be executed in the context of the loop function, using our single ops channel.

[click] In this case the signature of the operation is a function which takes a map of net.Addr's to net.Conn's. In a real program you'd probably have a much more complicated type to represent a client connection, but it's sufficient for the purpose of this example.

[click] remove is similar, we send a function that deletes it's connection's address from the supplied map

[click] send message is a function which iterates over all connections in the supplied map and calls write string to send each a copy of the message.

[click] You can see that we've moved the logic from the body of loop into anonymous functions created by our helpers.

So the job of loop is now to create a conns map, then waits for an operation to be provided on the ops channel, and invokes it, passing in the conns map.

But there are a few problems still to fix. The most pressing is the lack of error handling in SendMsg; an error writing to a connection will not be communicated back to the caller. So let's fix that now

```
func (m *Mux) SendMsg(msg string) error {
        m.ops <- func(m map[net.Addr]net.Conn) {
                for _, conn := range m {
                        io.WriteString(conn, msg)
                }
        }
        return nil
}

func (m *Mux) loop() {
        conns := make(map[net.Addr]net.Conn)
        for op := range m.ops {
                op(conns)
        }
}
```

Now, each method sends an operation to be executed in the context of the loop function, using our single ops channel.

[click] In this case the signature of the operation is a function which takes a map of net.Addr's to net.Conn's. In a real program you'd probably have a much more complicated type to represent a client connection, but it's sufficient for the purpose of this example.

[click] remove is similar, we send a function that deletes it's connection's address from the supplied map

[click] send message is a function which iterates over all connections in the supplied map and calls write string to send each a copy of the message.

[click] You can see that we've moved the logic from the body of loop into anonymous functions created by our helpers.

So the job of loop is now to create a conns map, then waits for an operation to be provided on the ops channel, and invokes it, passing in the conns map.

But there are a few problems still to fix. The most pressing is the lack of error handling in SendMsg; an error writing to a connection will not be communicated back to the caller. So let's fix that now

```
type Mux struct {
        ops chan func(map[net.Addr]net.Conn)
}

func (m *Mux) SendMsg(msg string) {
        result := make(chan error, 1)
        m.ops <- func(m map[net.Addr]net.Conn) {
                for _, conn := range m.conns {
                        err := io.WriteString(conn, msg)
                        if err != nil {
                                result <- err
                                return
                        }
                }
                result <- nil
        }
        return <-result
}
```

To handle the error being generated inside the anonymous function we pass to loop, we need to create a channel to communicate the result of the operation.

This also creates a point of synchronisation. The last line of SendMsg blocks until the function we passed into loop has been executed.

[click] Note that we didn’t have the change the body of loop at all to incorporate this error handling.

And now we know how to do this, we can easily add a new function to Mux to send a private message to a single client.

15:30

[click]

To do this we pass a "lookup function" to loop, which will look in the map provided to it—this is loop's conns map—and return the value for the address we want on the result channel

In the rest of the function we check to see if the result was nil—the zero value from doing the map lookup implies that the client is not registered. Otherwise we now have a reference to the client and we can call io.WriteString to send them a message.

And just to reiterate, we did this all without changing the body of loop, or affecting any of the other operations.

```
                for _, conn := range m.conns {
                        err := io.WriteString(conn, msg)
                        if err != nil {
                                result <- err
                                return
                        }
                }
                result <- nil
        }
        return <-result
}

func (m *Mux) loop() {
        conns := make(map[net.Addr]net.Conn)
        for op := range m.ops {
                op(conns)
        }
}
```

To handle the error being generated inside the anonymous function we pass to loop, we need to create a channel to communicate the result of the operation.

This also creates a point of synchronisation. The last line of SendMsg blocks until the function we passed into loop has been executed.

[click] Note that we didn’t have the change the body of loop at all to incorporate this error handling.

And now we know how to do this, we can easily add a new function to Mux to send a private message to a single client.

15:30

[click]

To do this we pass a "lookup function" to loop, which will look in the map provided to it—this is loop's conns map—and return the value for the address we want on the result channel

In the rest of the function we check to see if the result was nil—the zero value from doing the map lookup implies that the client is not registered. Otherwise we now have a reference to the client and we can call io.WriteString to send them a message.

And just to reiterate, we did this all without changing the body of loop, or affecting any of the other operations.

```
        for op := range m.ops {
                op(conns)
        }
}

func (m *Mux) PrivateMsg(addr net.Addr, msg string) error {
        result := make(chan net.Conn, 1)
        m.ops <- func(m map[net.Addr]net.Conn) {
                result <- m[addr]
        }

        conn := <-result
        if conn == nil {
                return errors.Errorf(
                        "client %v not registered", addr)
        }
        return io.WriteString(conn, msg)
}
```

To handle the error being generated inside the anonymous function we pass to loop, we need to create a channel to communicate the result of the operation.

This also creates a point of synchronisation. The last line of SendMsg blocks until the function we passed into loop has been executed.

[click] Note that we didn’t have the change the body of loop at all to incorporate this error handling.

And now we know how to do this, we can easily add a new function to Mux to send a private message to a single client.

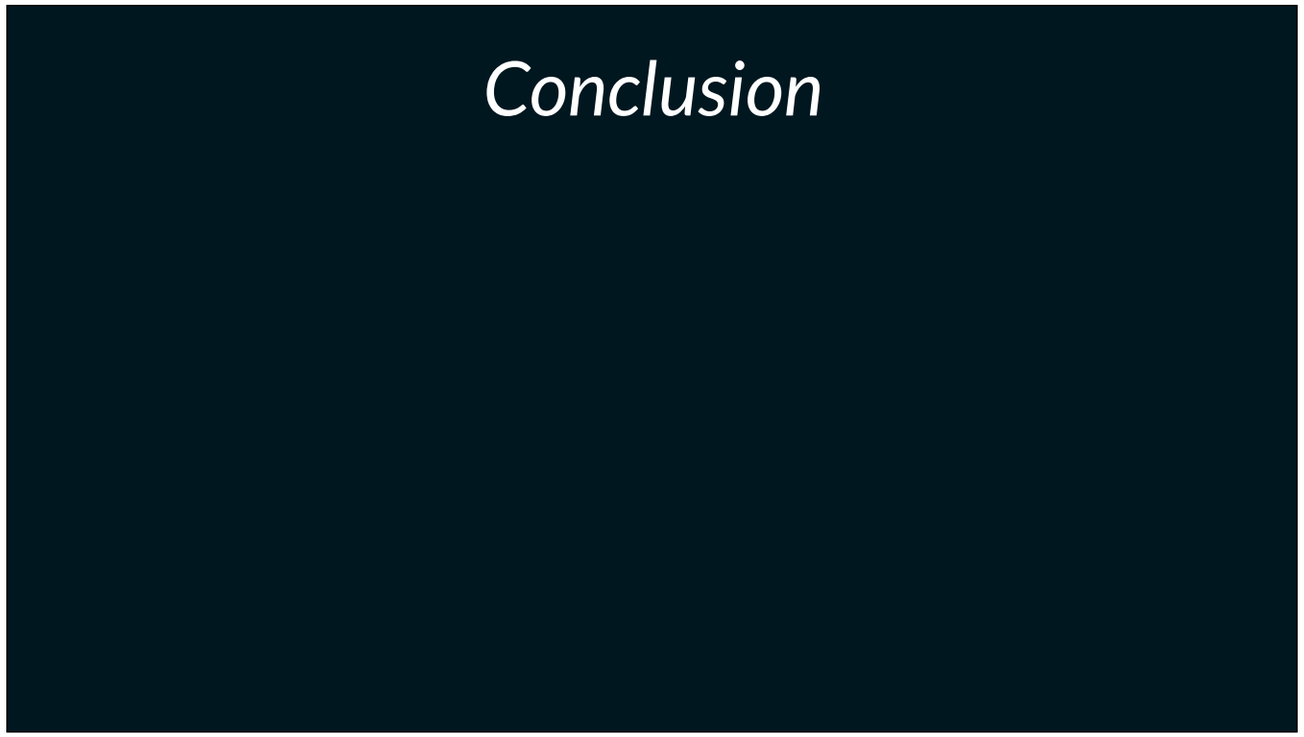
15:30

[click]

To do this we pass a "lookup function" to loop, which will look in the map provided to it—this is loop's conns map—and return the value for the address we want on the result channel

In the rest of the function we check to see if the result was nil—the zero value from doing the map lookup implies that the client is not registered. Otherwise we now have a reference to the client and we can call io.WriteString to send them a message.

And just to reiterate, we did this all without changing the body of loop, or affecting any of the other operations.



16:30

In summary

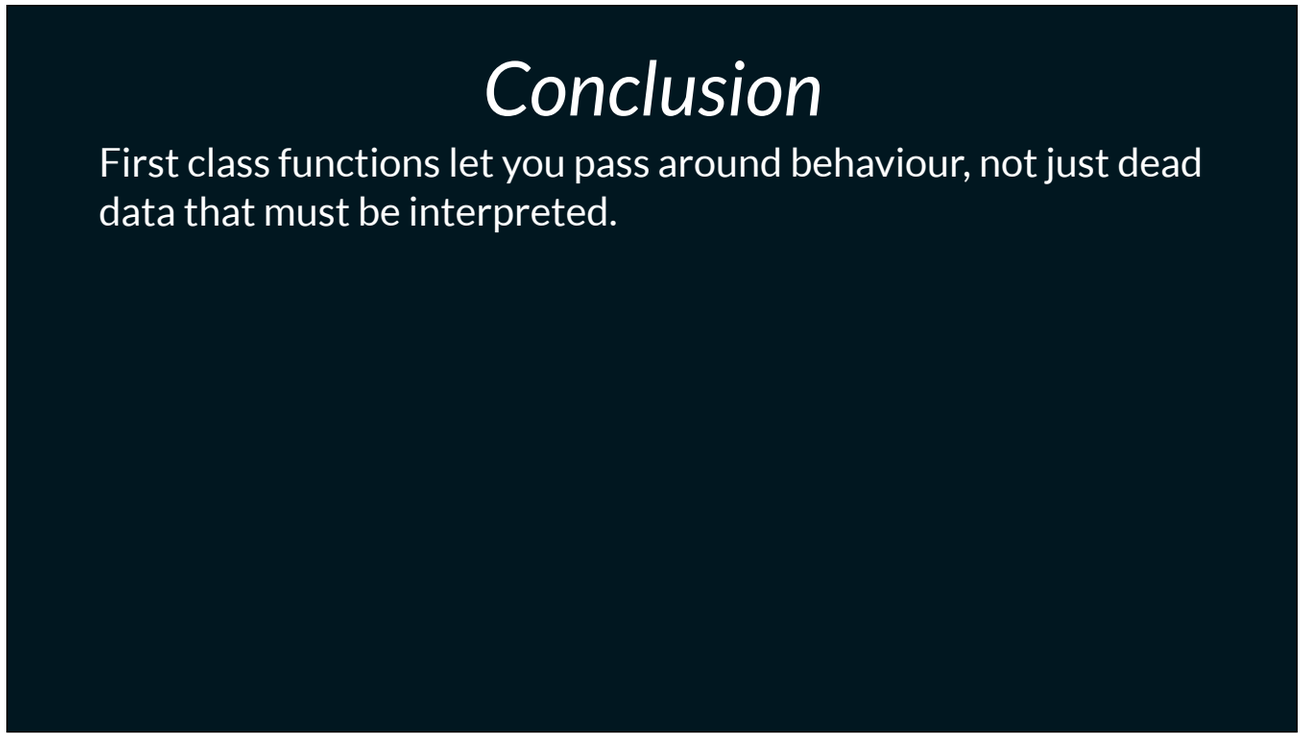
[click] First class functions bring us tremendous expressive power. They let you pass around behaviour, not just dead data that must be interpreted.

[click] First class functions aren't new or novel. Many older languages have offered them, even C, and in fact it was only somewhere along the lines of removing pointers did programmers in the OO stream of languages loose access to first class functions. If you're a Javascript programmer, you've probably spent the last 15 minutes wondering what the big deal is.

[click] First class functions, like the other features Go offers, should be used with restraint; just as it is possible to make an overcomplicated program with the overuse of channels, it's possible to make an impenetrable program with an overuse of first class functions. But that does not mean you shouldn't use them at all; just use them in moderation.

[click] First class functions are something that I believe every Go programmer should have in their toolbox. First class functions aren't unique to Go, and Go programmers shouldn't be afraid of them.

[click] If you can learn to use interfaces, you can learn to use first class functions. They aren't hard, just a little unfamiliar, and unfamiliarity is something that I believe can be overcome with time and practice.



16:30

In summary

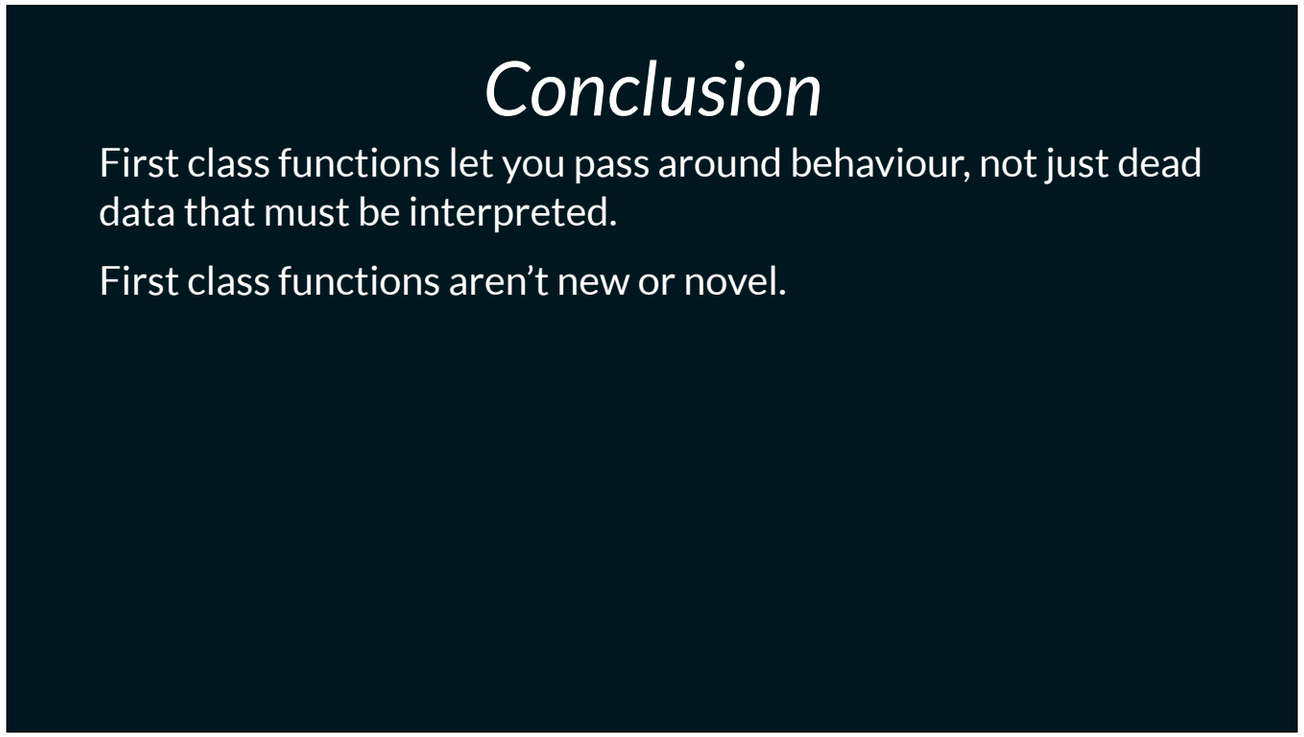
[click] First class functions bring us tremendous expressive power. They let you pass around behaviour, not just dead data that must be interpreted.

[click] First class functions aren't new or novel. Many older languages have offered them, even C, and in fact it was only somewhere along the lines of removing pointers did programmers in the OO stream of languages loose access to first class functions. If you're a Javascript programmer, you've probably spent the last 15 minutes wondering what the big deal is.

[click] First class functions, like the other features Go offers, should be used with restraint; just as it is possible to make an overcomplicated program with the overuse of channels, it's possible to make an impenetrable program with an overuse of first class functions. But that does not mean you shouldn't use them at all; just use them in moderation.

[click] First class functions are something that I believe every Go programmer should have in their toolbox. First class functions aren't unique to Go, and Go programmers shouldn't be afraid of them.

[click] If you can learn to use interfaces, you can learn to use first class functions. They aren't hard, just a little unfamiliar, and unfamiliarity is something that I believe can be overcome with time and practice.



16:30

In summary

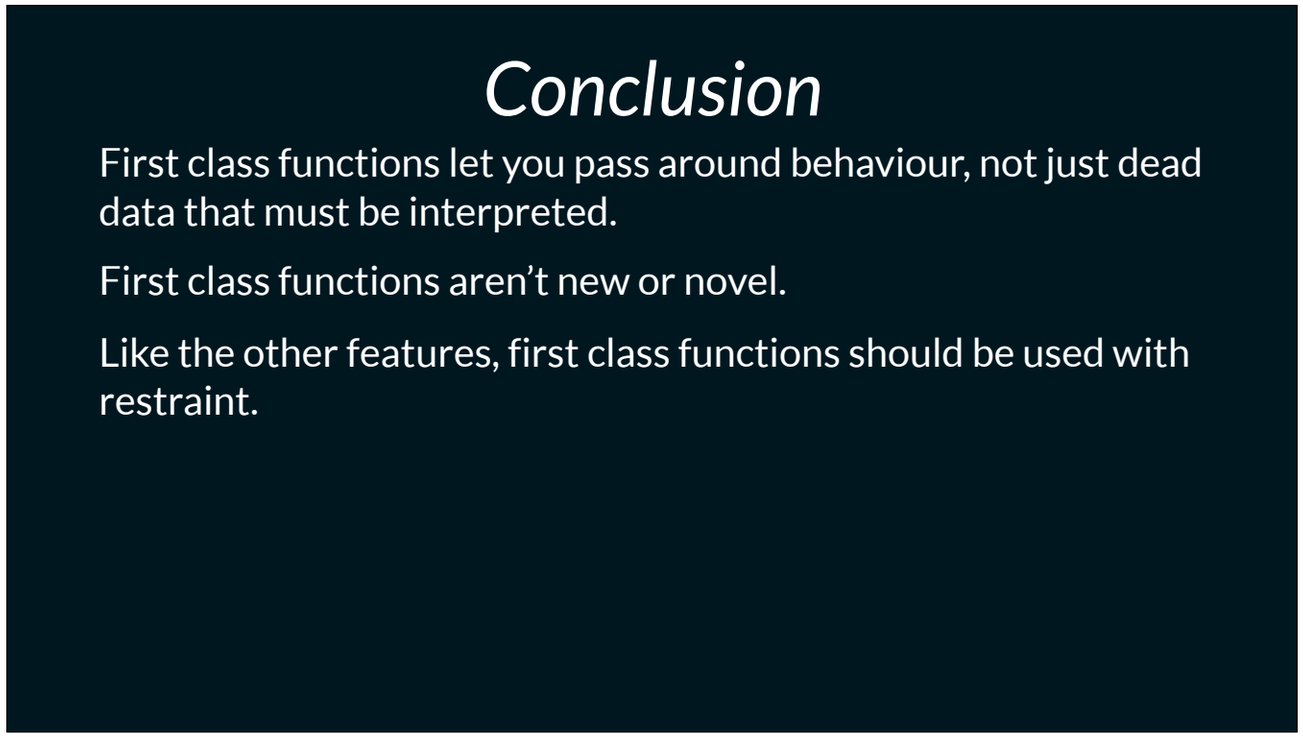
[click] First class functions bring us tremendous expressive power. They let you pass around behaviour, not just dead data that must be interpreted.

[click] First class functions aren’t new or novel. Many older languages have offered them, even C, and in fact it was only somewhere along the lines of removing pointers did programmers in the OO stream of languages loose access to first class functions. If you’re a Javascript programmer, you’ve probably spent the last 15 minutes wondering what the big deal is.

[click] First class functions, like the other features Go offers, should be used with restraint; just as it is possible to make an overcomplicated program with the overuse of channels, it’s possible to make an impenetrable program with an overuse of first class functions. But that does not mean you shouldn’t use them at all; just use them in moderation.

[click] First class functions are something that I believe every Go programmer should have in their toolbox. First class functions aren’t unique to Go, and Go programmers shouldn’t be afraid of them.

[click] If you can learn to use interfaces, you can learn to use first class functions. They aren’t hard, just a little unfamiliar, and unfamiliarity is something that I believe can be overcome with time and practice.



16:30

In summary

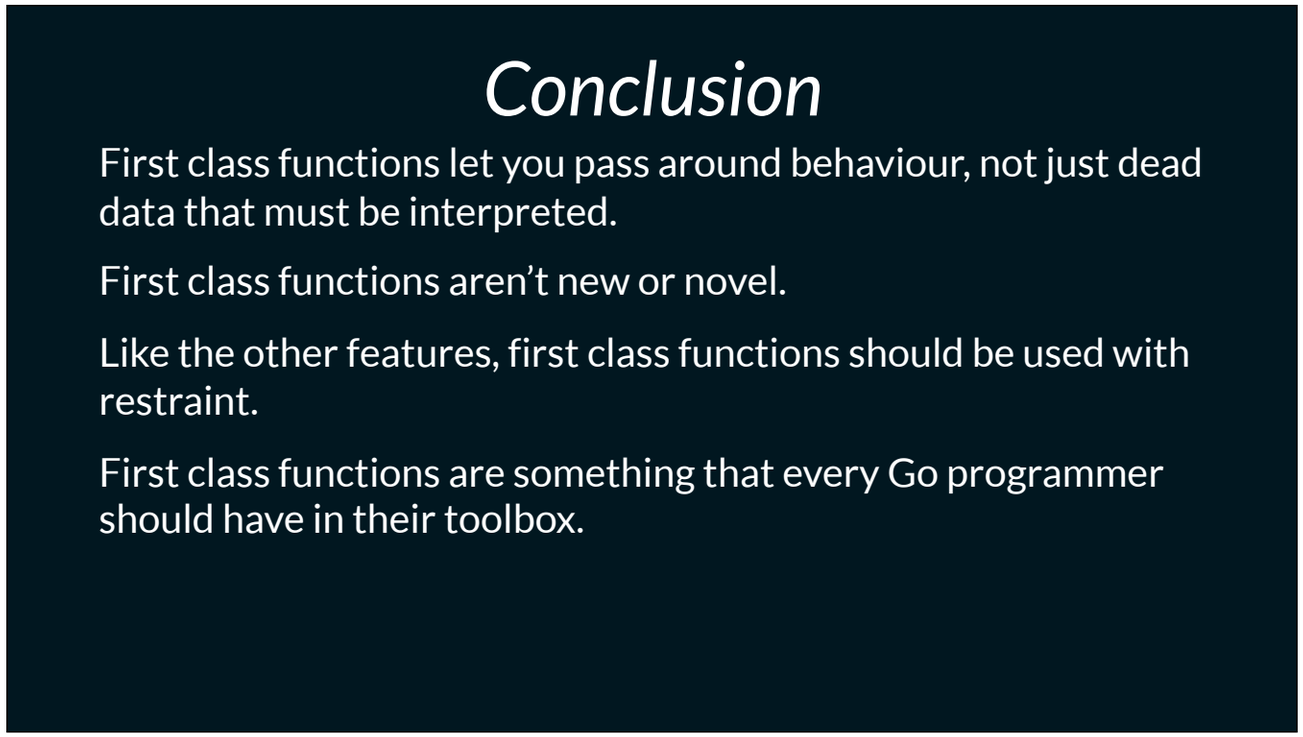
[click] First class functions bring us tremendous expressive power. They let you pass around behaviour, not just dead data that must be interpreted.

[click] First class functions aren’t new or novel. Many older languages have offered them, even C, and in fact it was only somewhere along the lines of removing pointers did programmers in the OO stream of languages loose access to first class functions. If you’re a Javascript programmer, you’ve probably spent the last 15 minutes wondering what the big deal is.

[click] First class functions, like the other features Go offers, should be used with restraint; just as it is possible to make an overcomplicated program with the overuse of channels, it’s possible to make an impenetrable program with an overuse of first class functions. But that does not mean you shouldn’t use them at all; just use them in moderation.

[click] First class functions are something that I believe every Go programmer should have in their toolbox. First class functions aren’t unique to Go, and Go programmers shouldn’t be afraid of them.

[click] If you can learn to use interfaces, you can learn to use first class functions. They aren’t hard, just a little unfamiliar, and unfamiliarity is something that I believe can be overcome with time and practice.



16:30

In summary

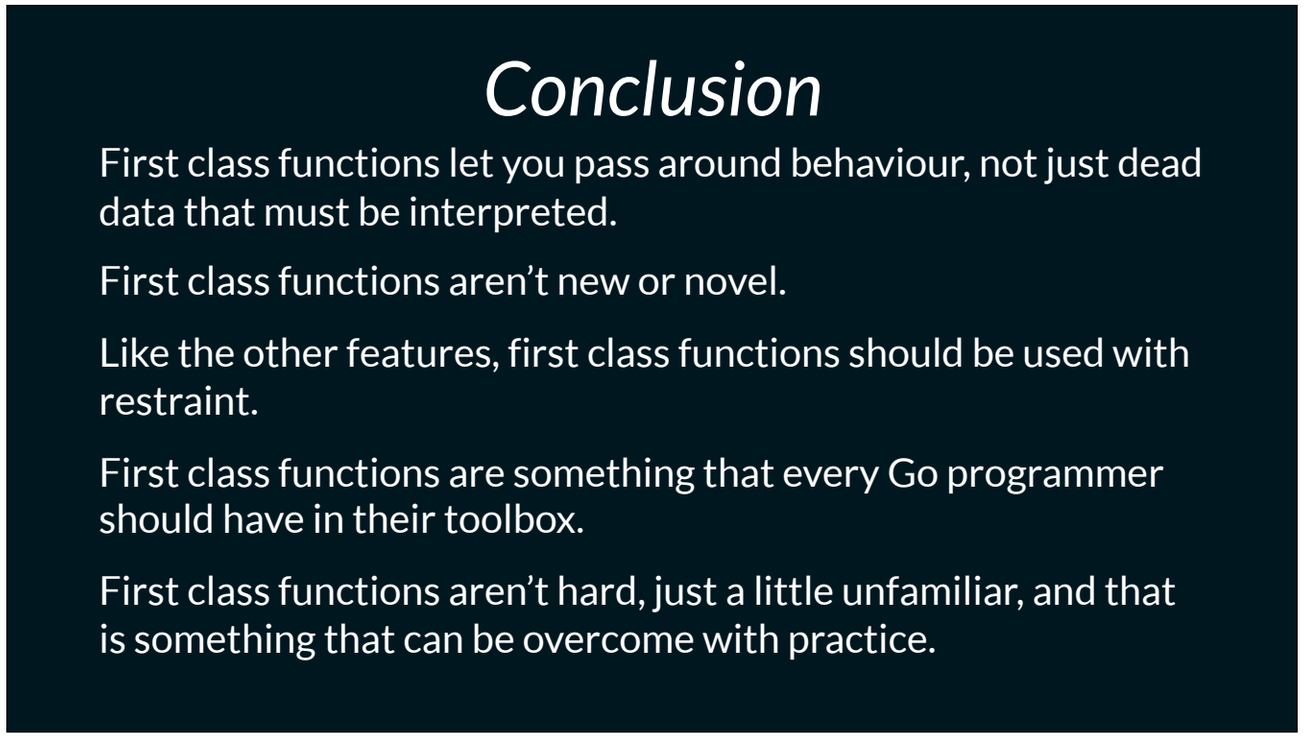
[click] First class functions bring us tremendous expressive power. They let you pass around behaviour, not just dead data that must be interpreted.

[click] First class functions aren’t new or novel. Many older languages have offered them, even C, and in fact it was only somewhere along the lines of removing pointers did programmers in the OO stream of languages loose access to first class functions. If you’re a Javascript programmer, you’ve probably spent the last 15 minutes wondering what the big deal is.

[click] First class functions, like the other features Go offers, should be used with restraint; just as it is possible to make an overcomplicated program with the overuse of channels, it’s possible to make an impenetrable program with an overuse of first class functions. But that does not mean you shouldn’t use them at all; just use them in moderation.

[click] First class functions are something that I believe every Go programmer should have in their toolbox. First class functions aren’t unique to Go, and Go programmers shouldn’t be afraid of them.

[click] If you can learn to use interfaces, you can learn to use first class functions. They aren’t hard, just a little unfamiliar, and unfamiliarity is something that I believe can be overcome with time and practice.



16:30

In summary

[click] First class functions bring us tremendous expressive power. They let you pass around behaviour, not just dead data that must be interpreted.

[click] First class functions aren’t new or novel. Many older languages have offered them, even C, and in fact it was only somewhere along the lines of removing pointers did programmers in the OO stream of languages loose access to first class functions. If you’re a Javascript programmer, you’ve probably spent the last 15 minutes wondering what the big deal is.

[click] First class functions, like the other features Go offers, should be used with restraint; just as it is possible to make an overcomplicated program with the overuse of channels, it’s possible to make an impenetrable program with an overuse of first class functions. But that does not mean you shouldn’t use them at all; just use them in moderation.

[click] First class functions are something that I believe every Go programmer should have in their toolbox. First class functions aren’t unique to Go, and Go programmers shouldn’t be afraid of them.

[click] If you can learn to use interfaces, you can learn to use first class functions. They aren’t hard, just a little unfamiliar, and unfamiliarity is something that I believe can be overcome with time and practice.

Thank you

@davecheney