Hi! I'm Adelina

Education Engineer at Spectro Cloud

Hi! I'm Adelina

You're in the book biz, kid!

Your company signs a new client - `vintagebooks.com`. 🥳 They carry the following types of books.

```
Novel: 1
Fairytale: 2
Drama: 3
Poetry: 4
Technical: 5
Biography: 6
```

You're in the book biz, kid!

`vintagebooks.com` is not your regular dusty bookstore! They also sell digital books.

```
Print: "Paperback", "Hardback"
Ebooks: "Kindle", "PDF"

Novel: 1
Fairytale: 2
Drama: 3
Poetry: 4
Technical: 5
Biography: 6
```

Refresher: Custom Types

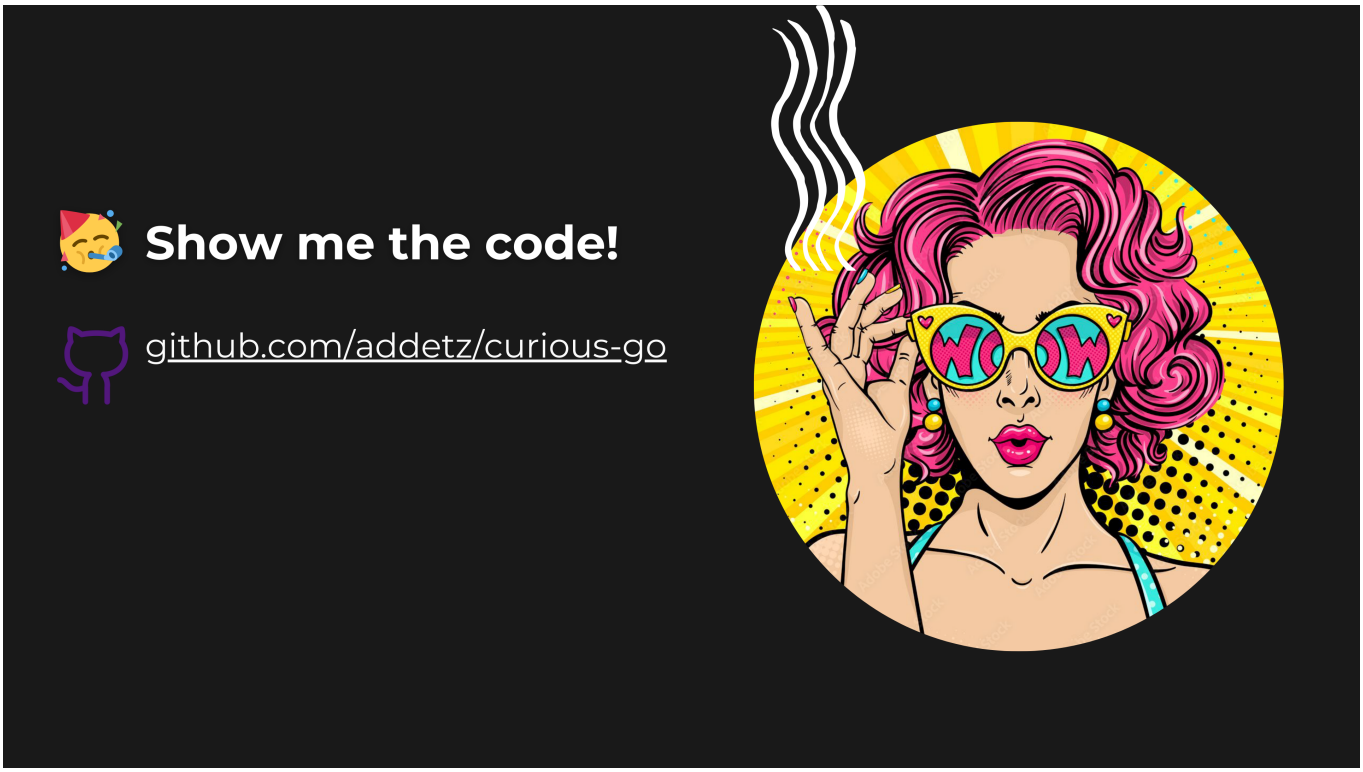
`structs` are typed collections of fields. Go does not have constructors, so we create our own function.

```
type MyType struct {
        MyField string
}

func NewMyType(value string) MyType {
        return MyType {
                MyField: value,
        }
}

func (m MyType) MyMethod() {
        // do something specific
}
```

Refresher: Enums

Enums in Go are represented by declaring constants of a custom type. `iota` is useful for incrementing values.

```
type MyNumber int

const (
        Zero MyNumber = iota
        One
        Two
)
```



Key Takeaways: Enums

We use **const** and custom types to build enums. They are zero-indexed, but their initial value is the **ordinal value** of their current const specification.

```
// iota is a predeclared identifier representing the untyped integer ordinal
// number of the current const specification in a (usually parenthesized)
// const declaration. It is zero-indexed.
const iota = 0 // Untyped int.
```

Key Takeaways: Enums

If enum values are used outside your code, you should declare these values explicitly.

```
// iota is a predeclared identifier representing the untyped integer ordinal
// number of the current const specification in a (usually parenthesized)
// const declaration. It is zero-indexed.
const iota = 0 // Untyped int.
```

The book biz never sleeps!

`vintagebooks.com` gives you a new assignment. Implement bookstore functionality for them, so they can begin to sort through their warehouse.

Requirements:

- Find a book in a performant way
- Add and maintain item count for a book

Refresher: Data Structures

Slices are dynamic arrays. They resize according to the memory requirements of their elements.

Syntax: `[]string`

Refresher: Data Structures

Maps are key-value stores. They use an underlying hash function to efficiently store and retrieve elements.

Syntax : `map[string]int`

Refresher: Pointers

Pointers are explicit in Go. The `&` operator generates a pointer to its operand. The pointer contains the memory address of the variable it's applied to.

```
// variable
i := 42
// pointer to variable
p := &i
```

Refresher: Pointers

The `*` returns the pointer's underlying value. This is known as *dereferencing* the pointer. The `*` and `&` operators are direct opposites.

```
// variable
i := 42
// pointer to variable
p := &i
// value of the pointer
j := *p
```

Refresher: Pointers

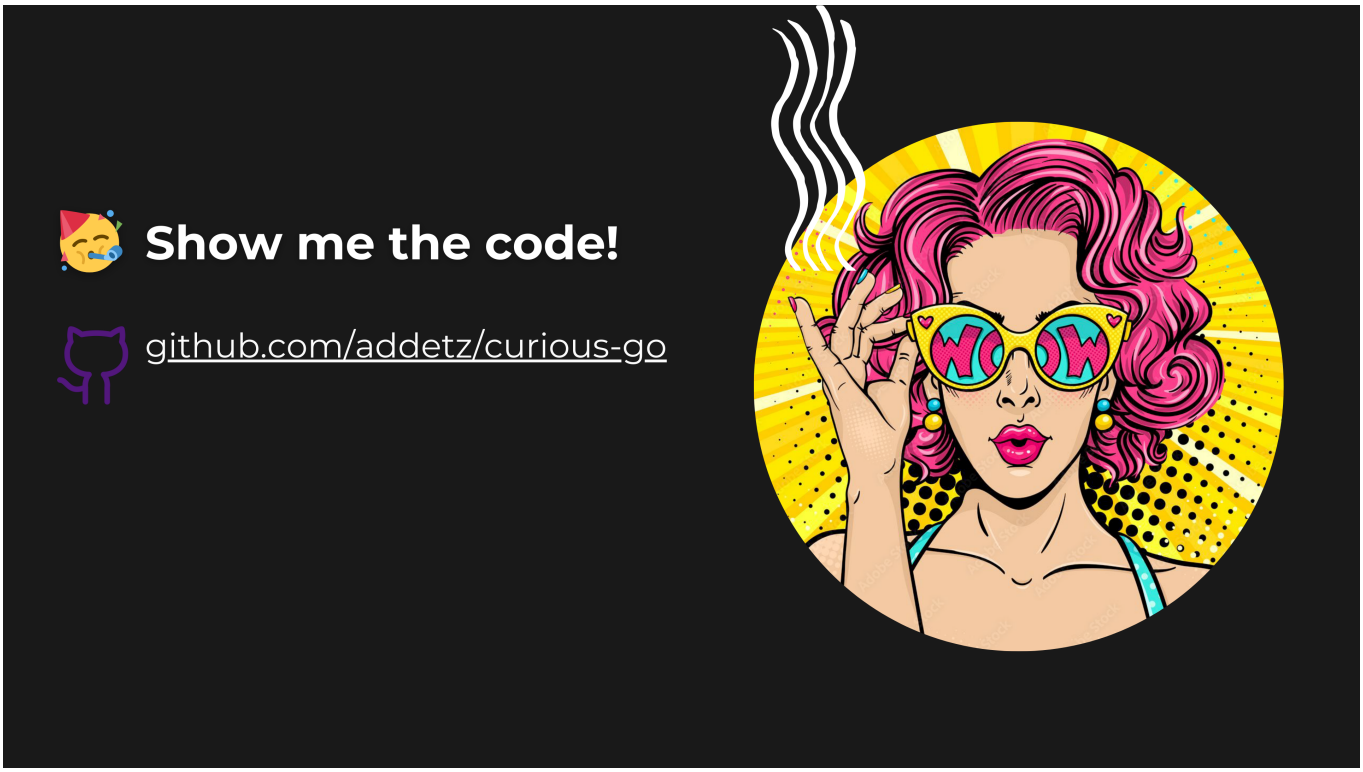
Unless specified using the pointer operator `*`, *most types* will be passed by value to functions and methods.

```
type MyType struct {
        value string
}

func (m MyType) NoModifications() {...}

func (m *MyType) Modifications() {...}
```

Refresher: Pointers

Slices and **Maps** are *reference types*. Their pointers are *implicit* and will always be passed by reference.

```

func Modifications(s []int) {...}

func Modifications(m map[string]int) {...}
```



Key Takeaways: Data Structures

Maps and slices are reference types. Even if the map or slice can be mutated, its elements may* still be passed by value.

* *unless the element types themselves are reference types*

Key Takeaways: Data Structures

It is best practice to use **immutable instances**. The principle of immutability leads to:

- Less side effects -> Robust code
- *Possibly* better performance
- Better encapsulation

The book biz is serious biz!

The next step is integrating with the client's external service. The books should be sent as JSON payloads.

```
{
 "title": "For Whom the Bell Tolls",
 "category": 1,
 "format": "Paperback"
}
```

Refresher: JSON Operations

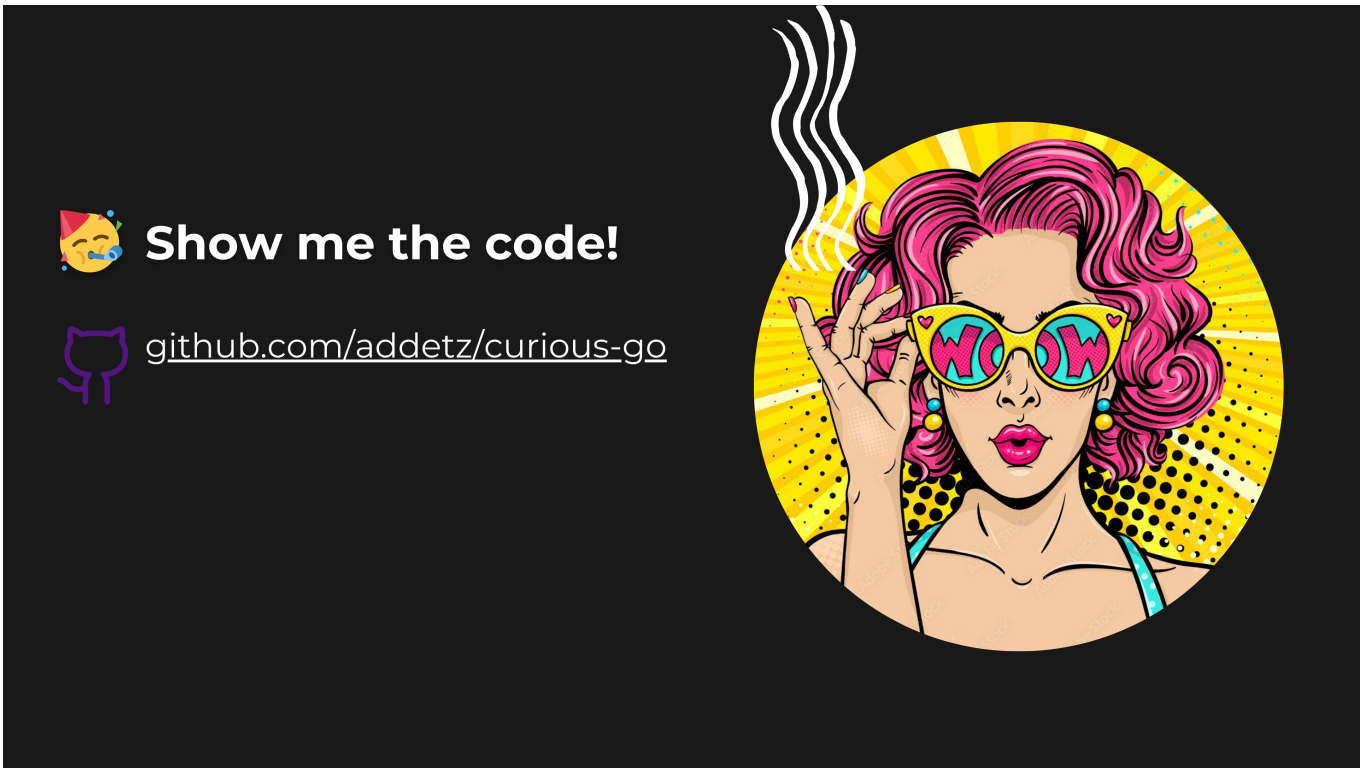
The Go standard library supports JSON operations through the `encoding/json` package.

```
// encode to JSON
func Marshal(v interface{}) ([]byte, error)

// decode JSON
func Unmarshal(data []byte, v interface{}) error
```

Refresher: JSON Operations

We use `json` tags to control the encoding. Otherwise, the struct field name will be used during encoding.

```
type Book struct {
        Title string `json:"title"`
        Category BookCategory `json:"category"`
        Format BookFormat `json:"format"`
}
```



Key Takeaways: JSON operations

JSON operations are part of the `encoding/json` package. By default, HTML escapes are set to true.

We can debate whether this makes sense, as JSON is widely used without HTML. 🤔

Key Takeaways: JSON Operations

Custom JSON encoders let you change the HTML escapes. They also allow you to undertake other advanced operations, such as custom time formats or data conversions/sanitization.

```go
// SetEscapeHTML specifies whether problematic HTML characters
// should be escaped inside JSON quoted strings.
// The default behavior is to escape &, <, and > to \u0026, \u003c, and \u003e
// to avoid certain safety problems that can arise when embedding JSON in HTML.
//
// In non-HTML settings where the escaping interferes with the readability
// of the output, SetEscapeHTML(false) disables this behavior.
func (enc *Encoder) SetEscapeHTML(on bool) {
    enc.escapeHTML = on
}
```

Thank You! 🫶

Slides: github.com/spectrocloud/go-guild/sessions
Code: github.com/addetz/curious-go

Please reach out with any feedback/suggestions. 🤓
