user defined types

main.go

```go
package main

import "fmt"

type person struct {
    name string
    age  int
}

func main() {
    p1 := person{"James", 20}
    fmt.Println(p1)
    fmt.Println(p1.name)
    fmt.Println(p1.age)
    fmt.Printf("%T\n",p1)
}
```

we already saw this code
when looking at structs

Terminal

```
01_struct $ go run main.go
{James 20}
James
20
main.person
01_struct $
```

user defined types

another example

main.go

```go
package main

import "fmt"

type mySentence string

func main() {
    var message mySentence = "Hello World!"
    fmt.Println(message)
    fmt.Printf("%T\n", message)
}
```

Terminal

```
02_string $ go run main.go
Hello World!
main.mySentence
02_string $
```

main.go

```go
package main

import "fmt"

type mySentence string

func main() {
    var message mySentence = "Hello World!"
    fmt.Println(message)
    fmt.Printf("%T\n", message)
    fmt.Printf("%T\n", string(message))
}
```

Terminal

```
03_string-conversion $ go run main.go
Hello World!
main.mySentence
string
03_string-conversion $
```

Conversion

main.go

```go
package main

import "fmt"

type mySentence string

func main() {
    var message mySentence = "Hello World!"
    fmt.Println(message)
    fmt.Printf("%T\n", message)
    fmt.Printf("%T\n", message.(string))
}
```

Assertion

Terminal

```
04_string_assertion_invalid-code $ go run main.go
# command-line-arguments
./main.go:11: invalid type assertion: message.(string) (non-interface type mySentence on left)
04_string_assertion_invalid-code $
```

type system

Pre-declared vs Composite

- Golang by default includes several **pre-declared, built-in, primitive** types
  - boolean
  - numeric
  - string
- Pre-declared types are used to construct other **composite** types
  - array
  - struct
  - pointer
  - slice
  - map
  - channel

main.go

```go
package main

import "fmt"

type person struct {
    name string
    age  int
}

func main() {
    var p1 person
    fmt.Println(p1)
    fmt.Println(p1.name)
    fmt.Println(p1.age)
    fmt.Printf("%T\n",p1)
}

// always use var to create and
// initialize a variable to its zero value
```

05_var-for-zero-val-initalization — bash

```
05_var-for-zero-val-initalization $ go run main.go
{ 0}

0
main.person
05_var-for-zero-val-initalization $ _
```

main.go

```go
package main

import "fmt"

type person struct {
    name string
    age  int
}

func main() {
    var p1 person
    fmt.Println(p1)
    fmt.Println(p1.name)
    fmt.Println(p1.age)
    fmt.Printf("%T\n",p1)
}

// always use var to create and
// initialize a variable to its zero value
```

Always use “var” to initialize a variable to its zero value

```
05_var-for-zero-val-initalization — bash
05_var-for-zero-val-initalization $ go run main.go
{ 0}

0
main.person
05_var-for-zero-val-initalization $ _
```

zero value

false for booleans, 0 for integers, 0.0 for floats, "" for strings
nil for pointers, functions, interfaces, slices, channels, and maps

https://golang.org/ref/spec#The_zero_value

main.go

```go
package main

import "fmt"

type person struct {
    name string
    age  int
}

func main() {
    p1 := person{
        name: "James",
        age:  20,
    }
    fmt.Println(p1)
    fmt.Println(p1.name)
    fmt.Println(p1.age)
    fmt.Printf("%T\n", p1)
}

// always use shorthand notation to
// create and initialize a variable to values
```

Always use shorthand notation to initialize a variable

```
06_shorthand-notation_nonzero-initalization — bash — 80×24
06_shorthand-notation_nonzero-initalization $ go run main.go
{James 20}
James
20
main.person
06_shorthand-notation_nonzero-initalization $ _
```

Named vs Unnamed

- Named
  - allow methods
- Unnamed / Anonymous

source: Learning Go - Types

Named vs Unnamed

```go
type Map map[string]string

//this is valid
func (m Map) Set(key string, value string) {
    m[key] = value
}

//this is invalid
func (m map[string]string)) Set(key string, value string) {
    m[key] = value
}
```

source: Learning Go - Types

golang spec

let’s continue learning about reading official documentation

Types

There are “named” and “unnamed” types

A type determines the set of values and operations specific to values of that type. Types may be *named* or *unnamed*. Named types are specified by a (possibly qualified) *type name*; unnamed types are specified using a *type literal*, which composes a new type from existing types.

```
Type      = TypeName | TypeLit | "(" Type ")" .
TypeName  = identifier | QualifiedIdent .
TypeLit   = ArrayType | StructType | PointerType | FunctionType | InterfaceType |
            SliceType | MapType | ChannelType .
```

Named instances of the boolean, numeric, and string types are predeclared. *Composite types*—array, struct, pointer, function, interface, slice, map, and channel types—may be constructed using type literals.

Each type `T` has an *underlying type*: If `T` is one of the predeclared boolean, numeric, or string types, or a type literal, the corresponding underlying type is `T` itself. Otherwise, `T`'s underlying type is the underlying type of the type to which `T` refers in its type declaration.

```
type T1 string
type T2 T1
type T3 []T1
type T4 T3
```

The underlying type of `string`, `T1`, and `T2` is `string`. The underlying type of `[]T1`, `T3`, and `T4` is `[]T1`.

https://golang.org/ref/spec#Types

Apps Bookmarks PM Hawk J Android GO

Types

“named” type - you create the identifier (the name)

A type determines the set of values and operations specific to values of that type. Types may be *named* or *unnamed*. Named types are specified by a (possibly qualified) *type name*; unnamed types are specified using a *type literal*, which composes a new type from existing types.

```
Type      = TypeName | TypeLit | "(" Type ")" .
TypeName  = identifier | QualifiedIdent .
```

Identifiers

Identifiers name program entities such as variables and types. An identifier is a sequence of one or more letters and digits. The first character in an identifier must be a letter.

```
identifier = letter { letter | unicode_digit } .
```

literal, the corresponding underlying type is T itself. Otherwise, T's underlying type is the underlying type of the type to which T refers in its type declaration.

```
type T1 string
type T2 T1
type T3 []T1
type T4 T3
```

The underlying type of `string`, `T1`, and `T2` is `string`. The underlying type of `[]T1`, `T3`, and `T4` is `[]T1`.

Types

"unnamed" type - use the TypeLit already defined

A type determines the set of values and operations specific to values of that type. Types may be *named* or *unnamed*. Named types are specified by a (possibly qualified) *type name*; unnamed types are specified using a *type literal*, which composes a new type from existing types.

```
Type      = TypeName | TypeLit | "(" Type ")" .
TypeName  = identifier | QualifiedIdent .
TypeLit   = ArrayType | StructType | PointerType | FunctionType | InterfaceType |
            SliceType | MapType | ChannelType .
```

Named insta
pointer, func

Each type T
literal, the c
type to whic

Slice types

A slice is a descriptor for a contiguous segment of an *underlying array* and provides access to a numbered sequence of elements from that array. A slice type denotes the set of all slices of arrays of its element type. The value of an uninitialized slice is `nil`.

```
SliceType = "[" "]" ElementType .
```

```
ElementType = Type .
```

```
type T1 string
type T2 T1
type T3 []T1
type T4 T3
```

The underlying type of `string`, `T1`, and `T2` is `string`. The underlying type of `[]T1`, `T3`, and `T4` is `[]T1`.

https://golang.org/ref/spec#Types

Apps Bookmarks

```
var results []int
fmt.Println(results)
```

Types

A type determines the set of values and operations specific to values of that type. Types may be *named* or *unnamed*. Named types are specified by a (possibly qualified) *type name*; unnamed types are specified using a *type literal*, which composes a new type from existing types.

```
Type      = TypeName | TypeLit | "(" Type ")" .
TypeName  = identifier | QualifiedIdent .
TypeLit   = ArrayType | StructType | PointerType | FunctionType | InterfaceType |
            SliceType | MapType | ChannelType .
```

Named insta
pointer, func

Each type T
literal, the c
type to whic

Slice types

A slice is a descriptor for a contiguous segment of an *underlying array* and provides access to a numbered sequence of elements from that array. A slice type denotes the set of all slices of arrays of its element type. The value of an uninitialized slice is `nil`.

```
SliceType = "[" "]" ElementType .
```

```
ElementType = Type .
```

```
type T1 string
type T2 T1
type T3 []T1
type T4 T3
```

```
mySlice := []int{1, 3, 5, 7, 9, 11,}
fmt.Printf("%T\n", mySlice)
```

```
01_int-slice $ go run main.go
[]int
```

The underlying type of `string`, `T1`, and `T2` is `string`. The underlying type of `[]T1`, `T3`, and `T4` is `[]T1`.

Types

There are “named” and “unnamed” types

A type determines the set of values and operations specific to values of that type. Types may be *named* or *unnamed*. Named types are specified by a (possibly qualified) *type name*; unnamed types are specified using a *type literal*, which composes a new type from existing types.

```
Type      = TypeName | TypeLit | "(" Type ")" .
TypeName  = identifier | QualifiedIdent .
TypeLit   = ArrayType | StructType | PointerType | FunctionType | InterfaceType |
            SliceType | MapType | ChannelType .
```

Named instances of the boolean, numeric, and string types are predeclared. *Composite types*—array, struct, pointer, function, interface, slice, map, and channel types—may be constructed using type literals.

Each type `T` has an *underlying type*: If `T` is one of the predeclared boolean, numeric, or string types, or a type literal, the corresponding underlying type is `T` itself. Otherwise, `T`'s underlying type is the underlying type of the type to which `T` refers in its type declaration.

```
type T1 string
type T2 T1
type T3 []T1
type T4 T3
```

The underlying type of `string`, `T1`, and `T2` is `string`. The underlying type of `[]T1`, `T3`, and `T4` is `[]T1`.

example

main.go

```go
package main

import "fmt"

type mySentences []string

func main() {
    var messages mySentences = []string{"Hello World!", "More coffee",}
    fmt.Println(messages)
    fmt.Printf("%T\n", messages)
}
```

Terminal

```
05_slice-strings $ go run main.go
[Hello World! More coffee]
main.mySentences
05_slice-strings $
```

main.go

```go
package main

import "fmt"

type mySentences []string

func main() {
    var messages mySentences = []string{"Hello World!", "More coffee",}
    fmt.Println(messages)
    fmt.Printf("%T\n", messages)
    fmt.Printf("%T\n", []string(messages))
}
```

Terminal

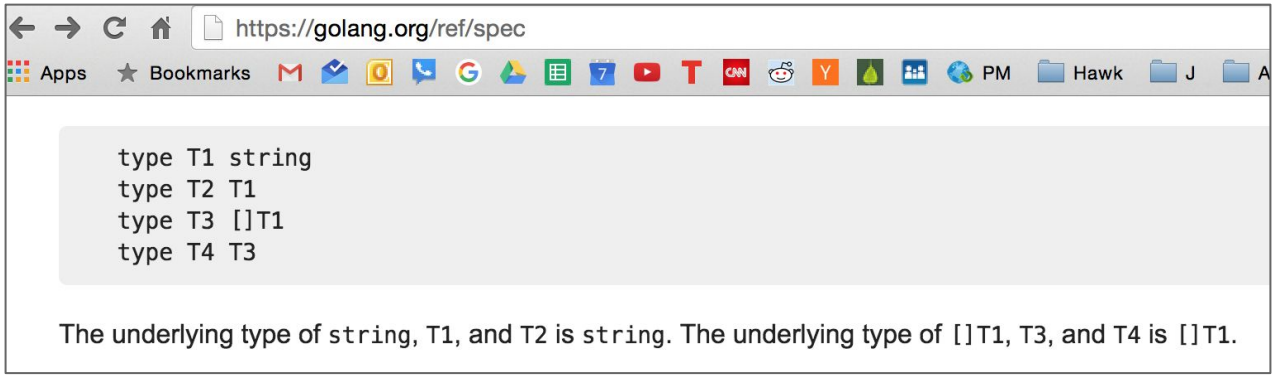
```
06_slice-strings_conversion $ go run main.go
[Hello World! More coffee]
main.mySentences
[]string
06_slice-strings_conversion $
```

Conversion

Underlying Type

- Every type has an underlying type
  - **Pre-declared** types and **type literals** refers to themselves as the underlying type.
  - When declaring a new type, you have to provide an existing type.
    - The new type will have the same underlying type as the existing type.

source: Learning Go - Types


https://golang.org/ref/spec
Apps
Bookmarks
PM
Hawk
J
type T1 string
type T2 T1
type T3 []T1
type T4 T3
The underlying type of string, T1, and T2 is string. The underlying type of []T1, T3, and T4 is []T1.

example

main.go

```go
package main

import "fmt"

type myType int

func main() {
    var x myType = 32
    fmt.Println(x)
    fmt.Printf("%T\n", x)
    fmt.Printf("%T\n", int(x))
}
```

```
Terminal
07_int $ go run main.go
32
main.myType
int
07_int $
```

Conversion

main.go

```go
package main

import "fmt"

type myType []int

func main() {
    var x myType = []int{32,44,57,}
    fmt.Println(x)
    fmt.Printf("%T\n", x)
    fmt.Printf("%T\n", []int(x))
}
```

Terminal

```
08_slice-ints $ go run main.go
[32 44 57]
main.myType
[]int
08_slice-ints $
```

Conversion

exercise

make a program
inside the program, create a type and use it

Review

- You can create your own types
  - your types will have an underlying type

Review Questions

predeclared

Why do you think the word “predeclared” is used when talking about certain types in go?

named

Why do you think the word “named” is used when talking about certain types in go?

composite

Why do you think the word “composite” is used when talking about certain types in go?

unnamed

Why do you think the word “unnamed” is used when talking about certain types in go?

literal

What is the general definition of a literal in programming? Why do you think the phrase “type literal” is used when talking about certain types in go?

named type

Write a program that creates a named typed, attaches a method to it, and then uses that method.