Overview of Data Types

This lesson tells us about the type of data that Go can handle.

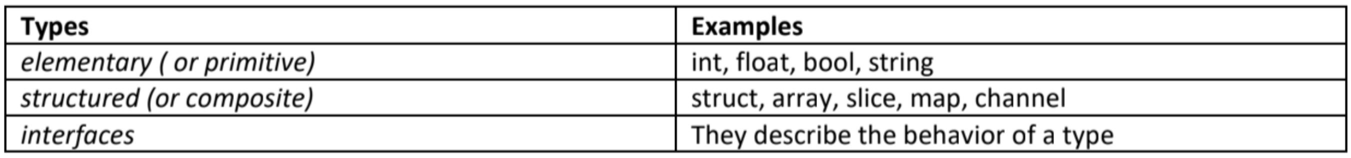
WE'LL COVER THE FOLLOWING

- Types
  - Conversions

Types

Variables contain data, and data can be of different *data types* or *types* for short. Go is a statically typed language. It means the compiler must know the types of all the variables, either because they were explicitly indicated, or because the compiler can infer the type from the code context. A type defines the set of values and the set of operations that can take place on those values. Here is an overview of some categories of types:

Types	Examples
elementary (or primitive)	`int`, `float`, `bool`, `string`
structured (or composite)	`struct`, `array`, `slice`, `map`, `channel`
interfaces	They describe the behavior of a type.



Types	**Examples**
elementary (or primitive)	int, float, bool, string
structured (or composite)	struct, array, slice, map, channel
interfaces	They describe the behavior of a type

Data Types in Go

A *structured* type, which has no real value (yet), has the value `nil`, which is also the *default* value for these types. To declare a variable, `var` keyword is used as:

```
var var1 type1
```

`var1` is the variable name, and `type1` is the type of `var1`.

Functions can also be of a certain type. The type of function is the type of variable which is returned by it. This *type* is written after the function name and its optional parameter-list, like:

```
func FunctionName (a typea, b typeb) typeFunc
```

So, you can see that `typeFunc` is the (return) type of the function, `FunctionName`. The returned variable `var` of type `typeFunc` appears somewhere in the function in the statement as:

```
return var
```

A function can have more than *one* return variables. In this case, the return-types are separated by comma(s) and surrounded by **( )**, like:

```
func FunctionName (a typea, b typeb) (t1 type1, t2 type2)
```

Two variables are returned with type `type1` and `type2` respectively. For such a case, the return statement takes the form:

```
return var1, var2
```

We can also have *user-defined* data types (our own data types), which we'll study in detail in Chapter 8. But, it is possible to have an alias for data types similar to what we have for packages. For example, to create an alias for an integer type you can do:

```
type IZ int
```

Now to declare an integer variable, we have to use an alias like:

Now to declare an integer variable, we have to use an alias like:

```
var a IZ = 5
```

If you have more than one type to define, you can use the factored keyword form, as in:

```
type (
  IZ int
  FZ float32
  STR string
)
```

In the above code, we create `IZ`, `FZ`, `STR` as an alias for `int`, `float32`, and `string` respectively.

Conversions

Sometimes a value needs to be converted into a value of another type called **type-casting**. Go does not allow implicit conversion, which means Go never does such a conversion by itself. The conversion must be done explicitly as **valueOfTypeB** = **typeB(valueOfTypeA)**. How about writing a program on type-casting and observing the result? Let's get started.

```
package main
import "fmt"

func main(){
    var number float32 = 5.2         // Declared a floating point variable
    fmt.Println(number)              // Printing the value of variable
    fmt.Println(int(number))         // Printing the type-castes result
}
```

Type Casting

At **line 5** we declare a floating-point variable called `number`, and print the value in the next line. Then at **line** 7 we print the type-casted value by converting the value to *int* type using `int(number)`. You can see that the value **5.2** becomes **5** after type-casting.

Now that you are familiar with data types. Let’s begin storing some data, in the next lesson.