Type Assertions vs Type Conversions in Golang

March 14, 2020 · Soham Kamani

Type assertions and type conversions seem to be a confusing topic in Go, since they both seem to do the same thing.

In this article, we will see how assertions and conversions are actually quite different, and go under the hood to see what happens when you use each of them in Go.

First, let's see what they look like…

This is a type assertion in Go:

```go
var greeting interface{} = "hello world"
greetingStr := greeting.(string)
```

And this is a type conversion:

```go
greeting := []byte("hello world")
greetingStr := string(greeting)
```

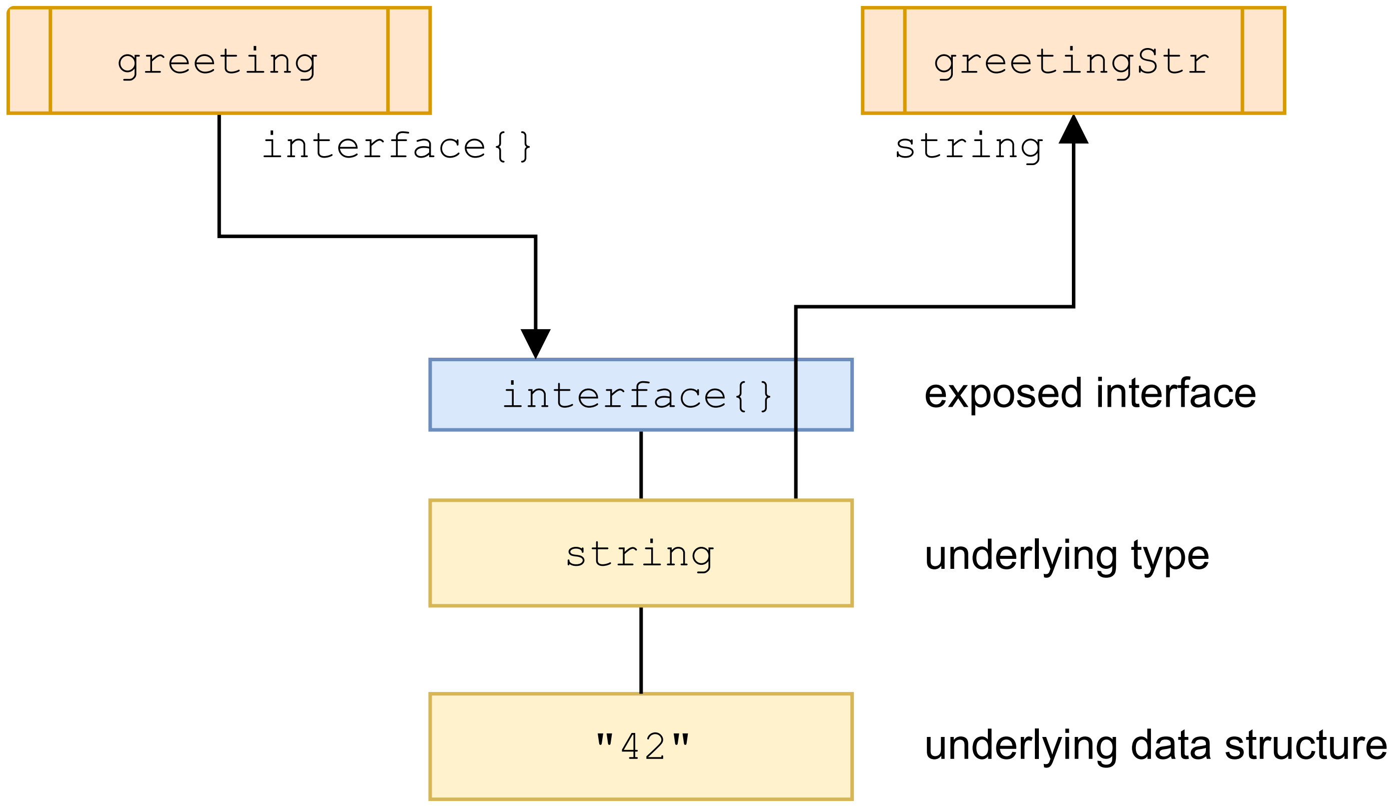
The most obvious difference is that they have a different syntax (`variable.(type)` vs `type(variable)`). Let's look at each case in detail.

Type Assertions

As the name suggests, type assertions are used to *assert* that a variable is of some type. Type assertions can *only take place on interfaces*.

In our type assertion example above, `greeting` was an `interface{}` type, to which we assigned a string. Now, we can say that `greeting` is actually a `string`, but the interface exposed to us is `interface{}`.

If we want to get the *original* type of `greeting` we can assert that it is a string, and this assertion returns its original `string` type.



This implies that we should know the underlying type of any variable while we are doing a type assertion, which is not always the case. This is why type assertion expressions actually return a second optional value:

```
var greeting interface{} = "42"
greetingStr, ok := greeting.(string)
```

The second value, `ok`, is a boolean value which is `true` if our assertion is correct, and false otherwise.

This also implies that type assertions are performed *at runtime*.

Type Switch

A type switch is a useful construct that you can use when you aren’t sure of the type of an interface:

```
var greeting interface{} = 42

switch g := greeting.(type) {
  case string:
    fmt.Println("g is a string with length", len(g))
  case int:
    fmt.Println("g is an integer, whose value is", g)
  default:
    fmt.Println("I don't know what g is")
}
```

Why Is It An Assertion?

In the above examples, it may seem like you’re “converting” the type of `greeting` from `interface{}` to `int` or `string`. However, the type of `greeting` is fixed, and is the same as what you declare during its initialization.

When you assign `greeting` to an interface type, you *do not change its underlying type*. In the same way, when you assert its type, you’re simply using the entire original types functionality, rather than the limited methods exposed by the interface.

Type Conversions

First, let's take a moment to understand what a "type" actually is. Each type in Go defines two things:

1. How the variable is stored (the underlying data structures)
2. What you can do with the variable (methods and functions it can be used in)

There are a few base types, of which `string` and `int` are included. And composite types, which include structs, maps, arrays and slices.

You can declare a new type from a base type, or by creating a composite type:

```
// myInt is a new type who's base type is `int`
type myInt int

// The AddOne method works on `myInt` types, but not regular `int`s
func (i myInt) AddOne() myInt { return i + 1}

func main() {
	var i myInt = 4
	fmt.Println(i.AddOne())
}
```

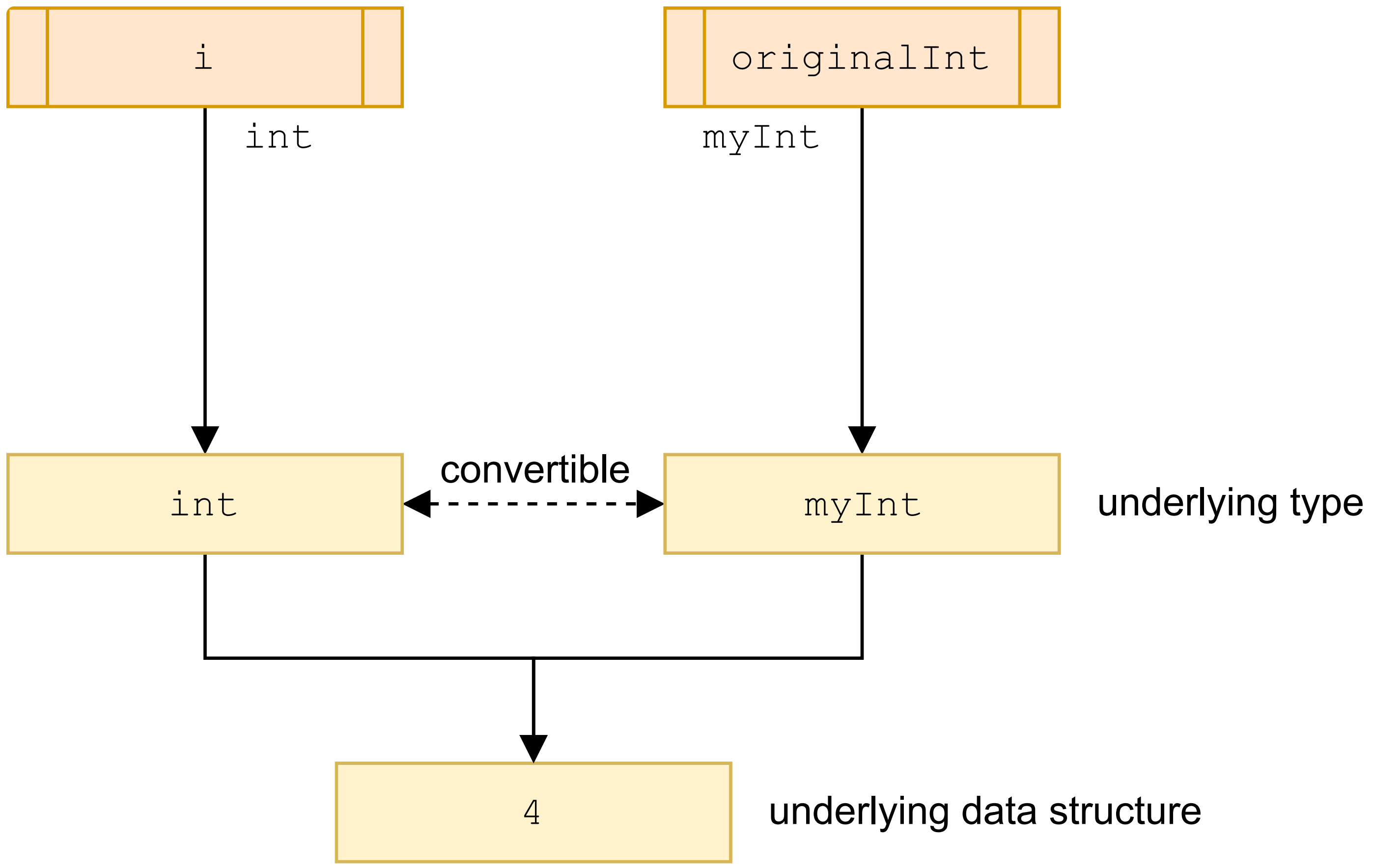
When we declared the `myInt` type, we based the variable data structure on the basic `int` type, but *changed* what we could do with `myInt` type variables (by declaring a new method on it).

Since the underlying data structure is similar for `int` and `myInt`, variables of these types can be converted between one another:

```
var i myInt = 4
```

```
originalInt := int(i)
```

Here `i` is of type `myInt`, but `originalInt` is of type `int`.



When Can We Use Type Conversion?

Type's can only be converted between one another *if* the underlying data structure is the same. Let's see an example using structs:

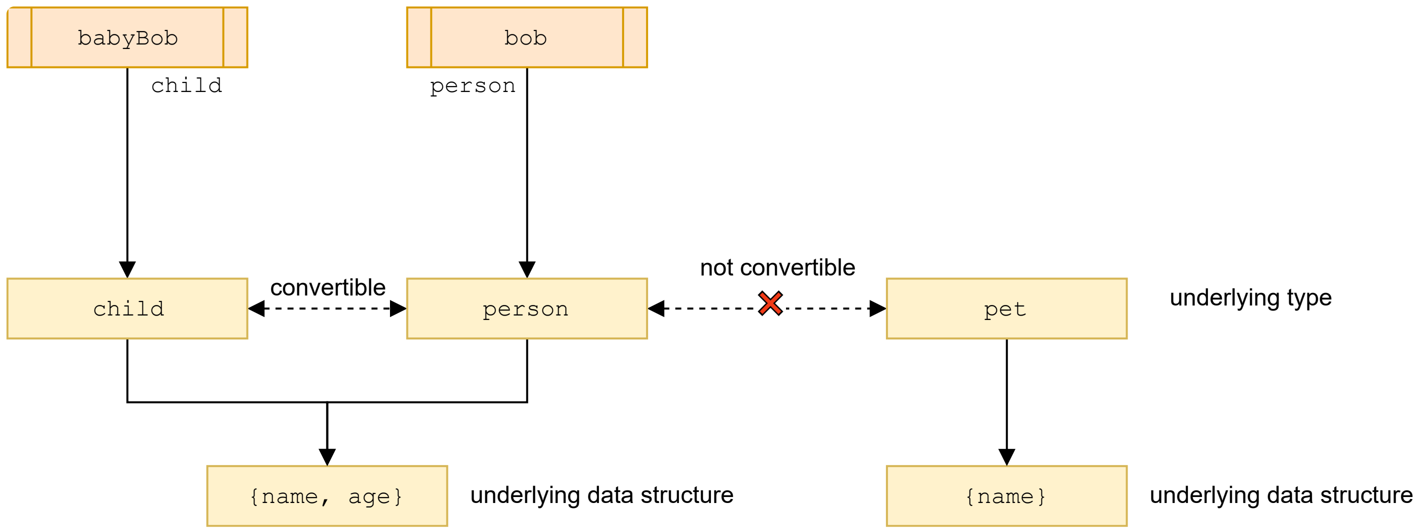
```
type person struct {
```

```go
    name string
    age int
}

type child struct {
    name string
    age int
}

type pet {
  name string
}

func main() {
    bob := person{
        name: "bob",
        age: 15,
        }
  babyBob := child(bob)
  // "babyBob := pet(bob)" would result in a compilation error
    fmt.Println(bob, babyBob)
}
```

Here, `person` and `child` have the same data structure, which is:

```go
struct {
    name string
    age int
}
```

They can therefore be converted between one another.



A shorthand can be used for declaring multiple types with the same data structure:

```
type pet person
```

This just means that `child` is based on the same data structure as `person` (similar to our integer example before)

Run this example here

Why Is It Called Conversion

As mentioned before, different types have different restrictions and methods defined on them, even though their data structure may be the same. When you convert from one type to another, you are *changing* what you can do with the type, rather than just *exposing* its underlying type, as is done in type assertions.

Type conversions also give you compilation errors if you try to convert to the wrong type, as opposed to runtime errors and optional `ok` return values that type assertions give.

Conclusion

The difference between type assertion and type conversion is more fundamental than just a difference in syntax. It also highlights the difference between interface types and non-interface (or concrete) types in Go.

An `interface` type does not have any underlying data structure, but rather exposes a few methods of a pre-existing concrete type (which does have an underlying data structure).

A type assertion brings out the concrete type underlying the interface, while type conversions change the way you can use a variable between two concrete types that have the same data structure.