Embedded Interface and Type Assertions

This lesson explains how embedding the interfaces works like a charm and how a function is called depending upon the type of interface decided at runtime.

WE'LL COVER THE FOLLOWING

- Interface embedding interface(s)
- Detecting and converting the type of an interface variable

Interface embedding interface(s)

An *interface* can contain the name of one (or more) interface(s), which is equivalent to explicitly enumerating the methods of the embedded interface in the containing interface. For example, the interface `File` contains all the methods of `ReadWrite` and `Lock`, in addition to a `Close()` method:

```
type ReadWrite interface {
  Read(b Buffer) bool
  Write(b Buffer) bool
}

type Lock interface {
  Lock()
  Unlock()
}

type File interface {
  ReadWrite
  Lock
  Close()
}
```

Detecting and converting the type of an interface variable

An interface type variable `varI` can contain a value of any type; we must have a means to detect this dynamic type, which is the actual type of the value stored in the variable at run time. The dynamic type may vary during execution but is always assignable to the type of the interface variable itself.

In general, we can test if `varI` contains at a certain moment a variable of type `T` with the type assertion test:

```
v := varI.(T) // unchecked type assertion
```

`varI` must be an *interface variable*. If not, the compiler signals the error: `invalid type assertion: varI.(T) (non-interface type (type of varI) on left)`

A type assertion may not be valid. The compiler does its utmost best to see if the conversion is valid, but it cannot foresee all possible cases. If this conversion fails while running the program, a runtime error occurs! A safer way is to use the following form:

```
if v, ok := varI.(T); ok { // checked type assertion
Process(v)
return
}
// here varI is not of type T
```

If this conversion is valid, `v` will contain the value of `varI` converted to type `T` and `ok` will be *true*. Otherwise, `v` is the zero value for `T` and `ok` is *false*, so no runtime error occurs!

> **Note:** Always use the comma, ok form for type assertions

In most cases, you would want to test the value of ok in an if. Then, it is most convenient to use the form:

```
if v, ok := varI.(T); ok {
// ...
}
```

In this form, shadowing the variable `varI` by giving `varI` and `v` the same name is sometimes done.

An example can be seen below:

```
package main
import (
"fmt"
"math"
)

type Square struct {
  side float32
}

type Circle struct {
  radius float32
}

type Shaper interface {
  Area() float32
}

func main() {
  var areaIntf Shaper
  sq1 := new(Square)
  sq1.side = 5
  areaIntf = sq1
  // Is Square the type of areaIntf ?
  if t, ok := areaIntf.(*Square); ok {
    fmt.Printf("The type of areaIntf is: %T\n", t)
  }
  if u, ok := areaIntf.(*Circle); ok {
    fmt.Printf("The type of areaIntf is: %T\n", u)
  } else {
    fmt.Println("areaIntf does not contain a variable of type Circle")
  }
}

func (sq *Square) Area() float32 {
  return sq.side * sq.side
}

func (ci *Circle) Area() float32 {
  return ci.radius * ci.radius * math.Pi
}
```

Type Assertions

In the above code, at **line** 7, we make a struct of type `Square` with one field `side` of type *float32*. Similarly, at **line 11**, we make a struct of type `Circle` with one field `radius` of type *float32*. At **line 15**, we make an interface `Shaper` with a single method `Area()`. At **line 35**, we define a method `Area()` that can with a single method `Area()`. At **line 35**, we define a method `Area()` that can

be called by a pointer to the `Square` type object. Again, at **line 39**, we define a method `Area()` that can be called by a pointer to the `Circle` type object.

Now, look at `main`. We create a `Shaper` variable `areaIntf` at **line 20**. In the next line, we make a `Square` variable `sq1` and assign it to `areaIntf`. At **line 25**, we check whether `areaIntf` contains a reference to the type `Square` using the *if* condition. If `ok` is *true*, control will transfer to **line 26**, and its type will be printed. Otherwise, control will transfer to **line 28**. At **line 28**, we are checking whether `areaIntf` contains a reference to the type `Circle` using the *if* condition.

If `ok` is *true*, control will transfer to **line 29**, and its type will be printed. Otherwise, control will transfer to **line 31**, and a message will be printed on the screen that such type doesn't exist.

> **Remark**: If we omit the * in `areaIntf.(*Square)`, we get the *compiler-error*: `impossible type assertion: areaIntf (type Shaper) cannot have dynamic type Square (missing Area method)`.

Now, that you know how to embed interfaces, in the next lesson, we'll try a combination of switch-cases control structure and interfaces.