Interfaces in Go

Session 10

Golang course by Exadel

03 Nov 2022

Sergio Kovtunenko
Lead backend developer, Exadel

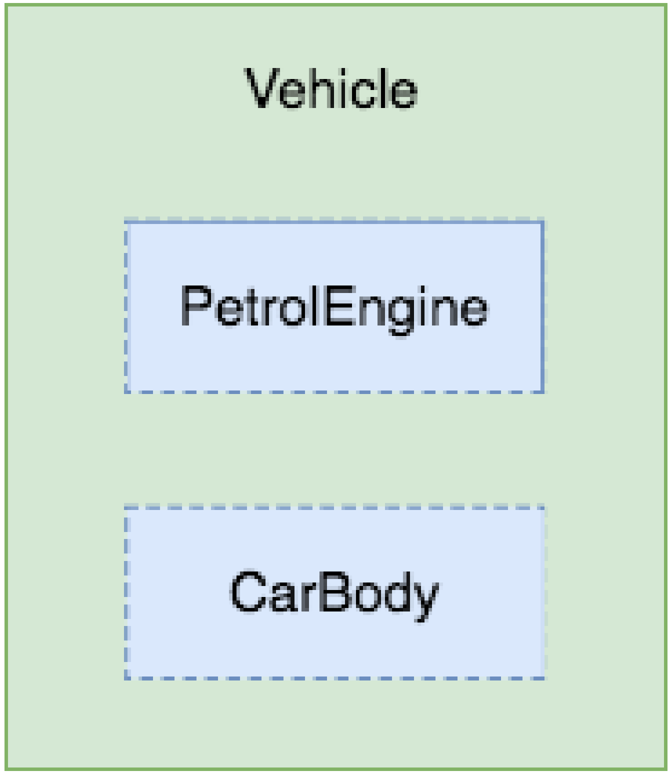
Agenda

- ▶️ Problem statement: why do we need interfaces?
- ▶️ Interface types
- ▶️ Empty Interface
- ▶️ Type assertions
- ▶️ Internals of interface types
- ▶️ Interface pitfalls and best practices
- ▶️ Next time...

Why do we need interfaces?

Let's model a vehicle... in a rigid way

▶️ Let's model a vehicle that can have only one type of engine and body:



Source: "5 things about programming I learned with Go" by MICHAŁ KONARSKI (http://mjk.space/5-things-about-programming-learned-with-go/)

Rigid modelling using struct types

▶️ Code at: `code/interfaceecample/01_system_without_interfaces_test.go`

Interface types

▶️ Abstract (no data!) 🤔

▶️ Define (possibly empty) set of method signatures 🤔

▶️ Values of *any_type* that implement all methods of an interface can be assigned to a variable of that interface.

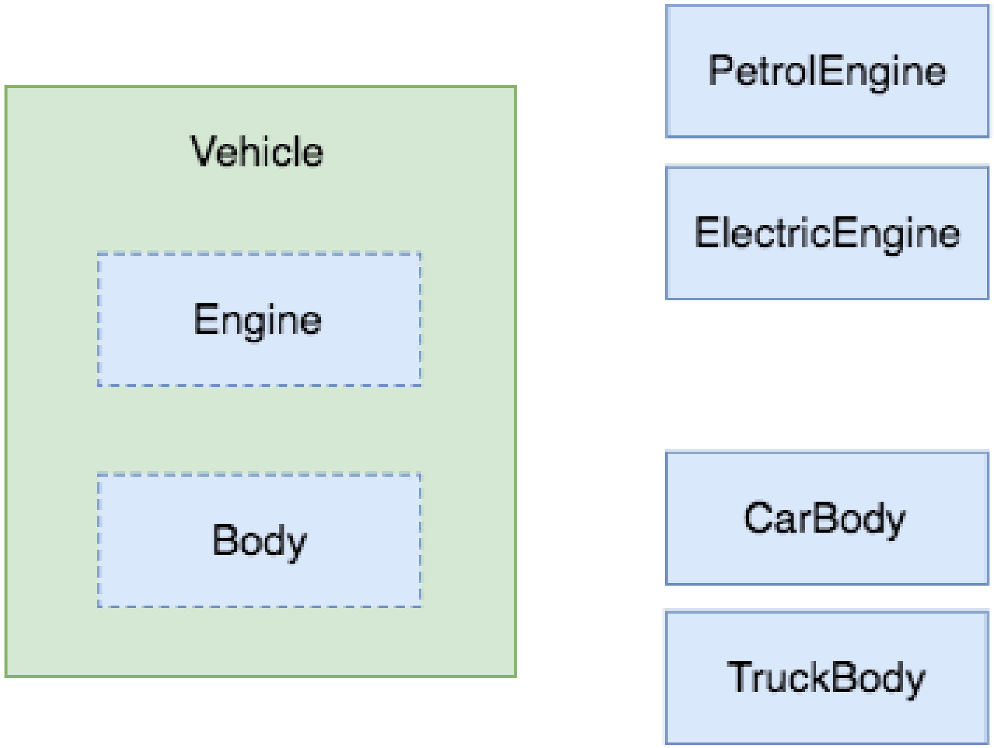
Examples:

```
type Anything interface{}  // empty interface

type Stringer interface {
    String() string
}

type Sorter interface {
    Len() int
    Swap(i, j int)
    Less(i, j int) bool
}
```

Source: "Introduction to Go" by Dave Chaney (https://github.com/davecheney/introduction-to-go)

Let's model a vehicle... in a flexible way

▶️ Let's model a vehicle that can have different types of engines and bodies:



Source: "5 things about programming I learned with Go" by MICHAŁ KONARSKI (http://mjk.space/5-things-about-programming-learned-with-go/)

Flexible modelling using interface types

▶️ Code at: `code/interfaceecample/02_system_with_interfaces_test.go`

What to choose?

▶️ Clear separation:

- Structures for data!
- Interfaces for behavior!

Interface types

Interfaces in Go: general information

- ✓ Why do we use interfaces?
  - Writing generic algorithms;
  - Hiding implementation details:
    - decouple implementation from API;
    - easily switch between implementations / or provide multiple ones;
  - Providing interception points.

- ✓ Main ideas behind **interfaces**:
  - Strict: interfaces for behaviour, static types for data.
  - *The broader interface, the weaker abstraction.*
  - Interface, in fact, should be created by consumer.
    - Define interfaces where you use them.
    - If you don’t want to provide multiple implementations of the same high-level behavior, you don’t introduce interfaces.
  - “A great rule of thumb for Go is ***accept interfaces, return structs.***” (Jack Lindamood)
    - Another advice: be generic when describing what a function needs, and be explicit when describing what a package provides.

- ✓ Having declared variable of interface type we know that:
  - There is *nothing real* about this variable.
  - There is *nothing concrete* about this variable.
  - This variable is **valueless**.

Interfaces in Go: rules to satisfy them

- ✓ Internally interfaces are **two words wide**:
  - schematically they look like: **(type, value)**
    - a pointer to a **method table** (holding type and method implementations)
    - a pointer to a **concrete value** (the type defined by the method table)
  - so values of interface types are *prone to race-conditions*. Because of 2-word nature.
- ✓ **Rule about implementing interfaces is simple**: “are the function's names and signatures exactly those of the interface?”.
- ✓ An interface can contain the name of one or more other interface(s), which is equivalent to *explicitly enumerating the methods of the embedded interface in the containing interface*.

Converting slices to interfaces can be done only in manual way

- ✓ Converting slices to interfaces can be done only in manual way:
  - because they *do not have* the same representation in memory.

```
t := []int{1, 2, 3, 4}
s := make([]interface{}, len(t))
for i, v := range t {
    s[i] = v
}
```

Calling a method on an interface value

- ✓ Calling a method *on an interface value* executes the method of the same name on its underlying type.
  - But calling a method on a nil interface is a *run-time error* because there is no type inside the interface tuple to indicate which concrete method to call.
- ✓ Whenever variables of any datatype is assigned to interface type, it is converted into interface type and stored.
  - So properties of *original data-type* cannot be retrieved until, it is converted again back to original data-type.
  - Conversion to data-type from interfaces *cannot be achieved using typecasting*, only type assertion.
- ✓ An interface value can also be assigned to another interface value, as long as the underlying value implements the necessary methods:

```go
// Values of interface `GetSet` can be assigned to values of type `Getter`.
type Getter interface{ Get() int }

// Values of interface `GetSet` can be assigned to values of type `Setter`.
type Setter interface{ Set(val int) }

// Values of interfaces `Getter` or `Setter` CAN NOT be assigned to values of type `GetSet`.
type GetSet interface {
    Getter
    Setter
}
```

Empty Interface

Empty Interface

- ✓ **Empty interface** Analogue to void* from C world or Object class from Java.
- ✓ The interface type that specifies zero methods is known as the empty interface.
- ✓ Empty interface says nothing:

```go
package main
import "fmt"

func main() {
    var i interface{}
    describe(i)

    i = 42
    describe(i)

    i = "hello"
    describe(i)
}

func describe(i interface{}) {
    fmt.Printf("(%v, %T)\n", i, i)
}
// OUTPUT:
// (<nil>, <nil>)
// (42, int)
// (hello, string)
```

- ✓ An *empty interface* may hold values of **any type** (Every type implements at least zero methods).
  - An **interface{}** value is not of any type; it is of **interface{}** type!

Empty Interface - code example

▶️ Code at: `code/interfaceecample/03_empty_interface_test.go`

Type assertions

Type assertions - general info

✓ A **type assertion** provides access to an **interface value's** underlying concrete value.

✓ In the `x.(T)` expression if the type `T`:

- *is not an interface type*, `x.(T)` asserts that the dynamic type of x is identical to the type T.
- *is an interface type*, `x.(T)` asserts that the dynamic type of x implements the interface T.

Type assertions - usecase

- ✓ If you have a value of interface type and want to convert it to another or a specific type (in case of *interface{}*), you can use type assertion.
  - A **type assertion** takes a value and tries to create another version in the specified explicit type.

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

func timeMap(y interface{}) {
    z, ok := y.(map[string]interface{})
    if ok {
        z["updated_at"] = time.Now()
    }
}

func main() {
    foo := map[string]interface{}{
        "Matt": 42,
    }
    timeMap(foo)
    fmt.Println(foo)
}
```

Type assertions - example

▶️ Code at: `code/interfaceecample/04_type_assert_test.go`

Type assertions - two forms

- ✓ There are two forms:
  - Form #1:

```
t := i.(T) // if something was wrong then panic!
```

    - This statement asserts that the interface value **i** holds the concrete type **T** and assigns the underlying **T** value to the variable **t**.
    - If **i** does not hold a **T**, the statement will trigger a panic.
  - Form #2:

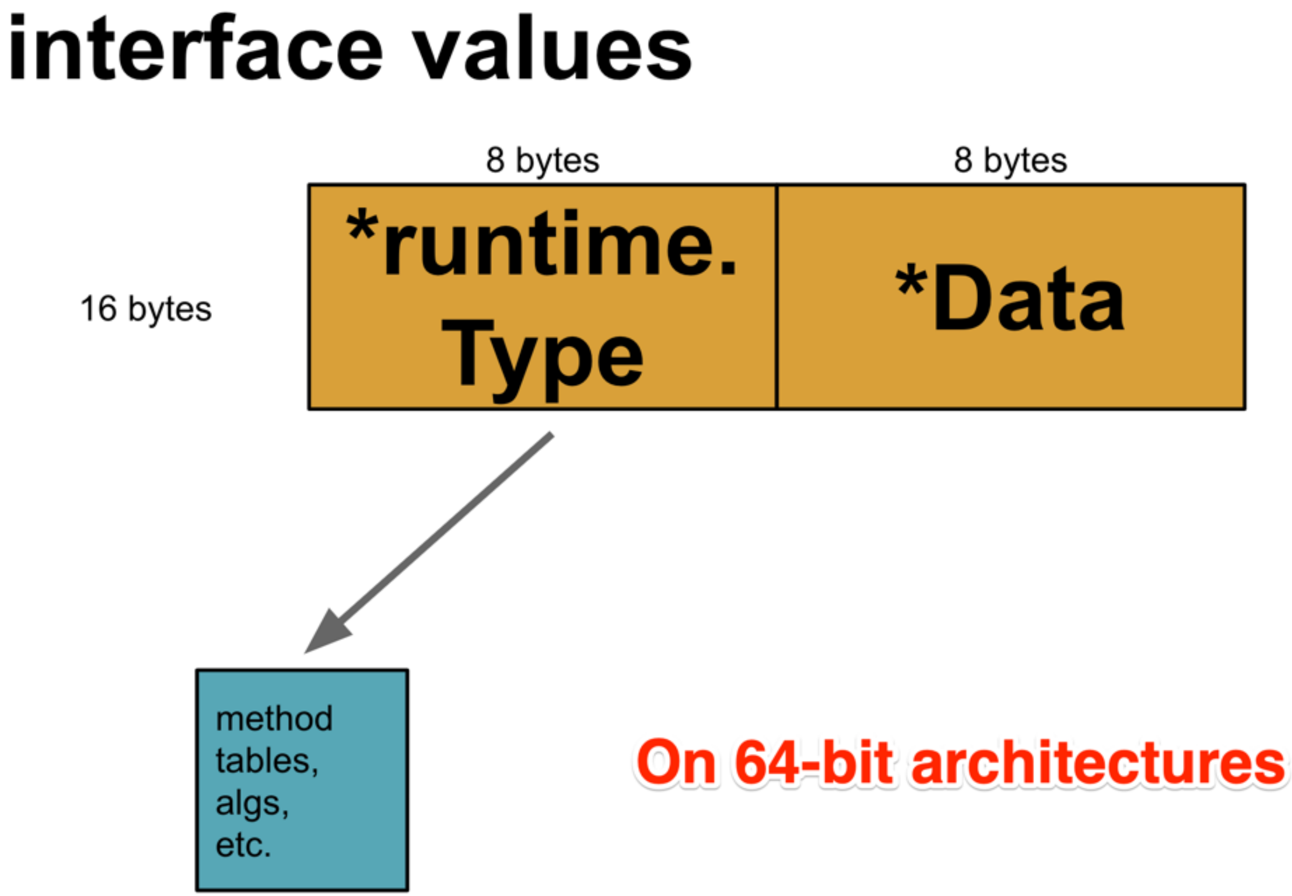
```
t, ok := i.(T) // no need to panic!
```

    - To test whether an interface value holds a specific type, a type assertion can return two values: the underlying value and a boolean value that reports whether the assertion succeeded.
      - If **i** holds a **T**, then **t** will be the underlying value and **ok** will be ***true***.
      - If not, **ok** will be *false* and **t** will be the zero value of type **T**, and no panic occurs.

- ✓ The type assertion doesn't have to be done on an empty interface.
  - It's often used when you have a function taking a param of a specific interface but the function inner code behaves differently based on the actual object type.

Internals of interface types

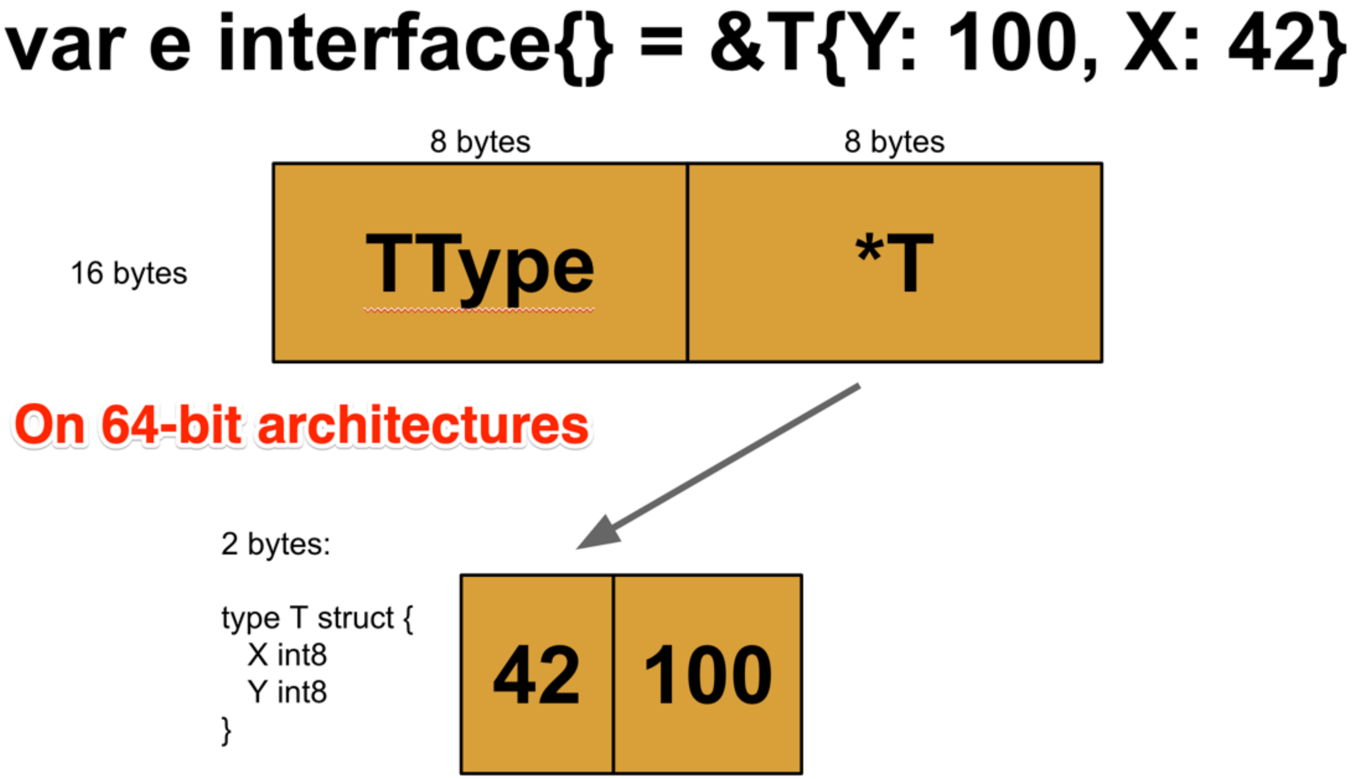
Interface internal representation in memory



Source: "Go Debugging, Profiling, and Optimization" by Brad Fitzpatrick

(https://docs.google.com/presentation/d/1lL7Wlh9GBtTSieqHGJ5AUd1XVYR48UPhEloVem-79mA/preview?sle=true&slide=id.p)

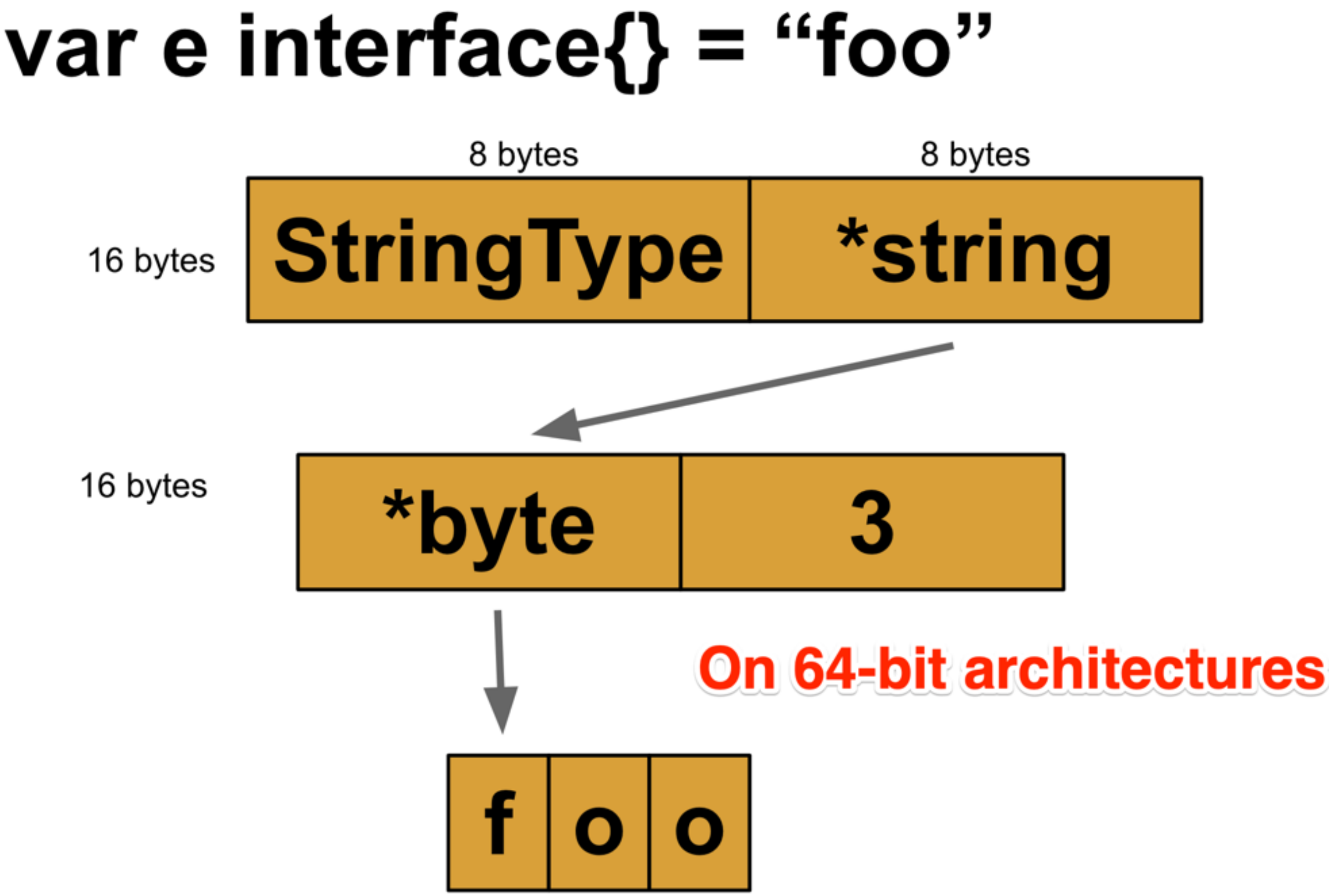
Interface holds a value



Source: "Go Debugging, Profiling, and Optimization" by Brad Fitzpatrick

(https://docs.google.com/presentation/d/1lL7Wlh9GBtTSieqHGJ5AUd1XVYR48UPhEloVem-79mA/preview?sle=true&slide=id.p)

Interface holds a string value



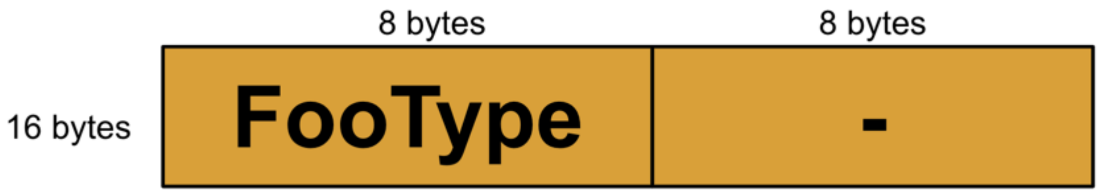
Source: "Go Debugging, Profiling, and Optimization" by Brad Fitzpatrick

(https://docs.google.com/presentation/d/1lL7Wlh9GBtTSieqHGJ5AUd1XVYR48UPhEloVem-79mA/preview?sle=true&slide=id.p)

Interface holds an empty struct value

```
type Foo struct{}  // 0-sized
var e interface{} = Foo{}
```

On 64-bit architectures



Source: "Go Debugging, Profiling, and Optimization" by Brad Fitzpatrick

(https://docs.google.com/presentation/d/1lL7Wlh9GBtTSieqHGJ5AUd1XVYR48UPhEloVem-79mA/preview?sle=true&slide=id.p)

Interface pitfalls and best practices

Interface pitfalls: nil receiver in methods

- ✓ **nil** receiver in methods:
  - If the *concrete* value inside the interface itself is **nil**, the method will be called with a nil receiver.
  - In some languages this would trigger a **null pointer exception**, but in Go it is common to write methods that gracefully handle being called with a nil receiver (as with the method M() in this example.)

▶️ Code at: `code/interfaceecample/05_nil_receiver_in_methods_test.go`

Interface pitfalls: nil receiver in methods - code

```go
package main
import "fmt"

func describe(i I) {
	fmt.Printf("(%v, %T)\n", i, i)
}

type I interface {
	M()
}

type T struct {
	S string
}

func (t *T) M() {
	if t == nil {
		fmt.Println("Got <nil> receiver")
		return
	}
	fmt.Println(t.S)
}

func main() {
	var i I

	var t *T // Not initialised!
	i = t
	describe(i)
	i.M() // nil receiver
	t.M() // nil receiver
}
// OUTPUT:
// (<nil>, *main.T)
// Got <nil> receiver
// Got <nil> receiver
```

How to guarantee that type satisfies an interface?

- ✓ Note that an interface value that holds a nil *concrete value* is itself **non-nil**.
- ✓ How to guarantee that type satisfies an interface?
  - Perform compiler check at *compile-time*:

```go
type I interface{}
type T struct{}
var _ I = T{}       // Verify that T implements I.
var _ I = (*T)(nil) // Verify that *T implements I.
```

  - Explicitly declare that type implements interface:
    - Most code doesn't make use of such constraints (sometimes, though, they're necessary to resolve ambiguities among similar interfaces)

```go
type Fooer interface {
    Foo()
    ImplementsFooer()
}
type Bar struct{}
func (b Bar) ImplementsFooer() {} // clearly documenting the fact and announcing it in godoc's output
func (b Bar) Foo() {}
```

Interface pitfalls: non-nil errors

- ▶️ This was discussed on the session 05: error handling
- ▶️ Code at: `code/interfaceecample/06_wrong_errors_check_test.go`

Interface pitfalls: failed type assertion

✓ Failed type assertion:

```
func main() {
    var data interface{} = "great" // actually a string value

    if res, ok := data.(int); ok {
        fmt.Println("[is an int] value =>", res)
    } else {
        fmt.Println("[not an int] value =>", res) // This is a BUG!
        // Failed type assertions return the "zero value" for the target type used in the assertion
statement:
        // prints: [not an int] value => 0 (not "great")

        fmt.Println("[not an int] value =>", data) // OK!
        // prints: [not an int] value => great (as expected)
    }
}
```

▶️ Code at: `code/interfaceecample/07_failed_type_assertion_test.go`

Interface best practice

- ✓ Don’t export any interfaces unless you have to encourage external packages to implement one.
- ✓ `io` package is a good starting point to study some of the the best practices.
  - It exports interfaces because it also needs to export generic-use functions like:
    `func Copy(dst Writer, src Reader) (written int64, err error)`.
- ✓ **Best practice:** Should your package export generic functionality? If the answer is a “*maybe*”, you’re likely to be polluting your package with an interface declaration.

Homework

Homework

▶️ Video: "Profiling & Optimizing in Go / Brad Fitzpatrick" (https://www.youtube.com/watch?v=xxDZuPEgbBU)

- Slides: "Go Debugging, Profiling, and Optimization" by Brad Fitzpatrick (https://docs.google.com/presentation/d/1lL7Wlh9GBtTSieqHGJ5AUd1XVYR48UPhEloVem-79mA/preview?sle=true&slide=id.p)

▶️ Video: Gopherfest 2015 | Go Proverbs with Rob Pike (https://www.youtube.com/watch?v=PAAkCSZUG1c&t=7m36s)

▶️ Slides: "Understanding the interface" by Francesc Campoy (https://speakerdeck.com/campoy/understanding-the-interface)

Next time...

▶️ Session11:

Testing in Go. Memory allocations and alignment

- Basic Tests, Benchmarks, Table-driven tests, Main test, Mocking, testify assertion library
- Allocation on stack vs. on the heap, Escape analysis, Memory alignment

Thank you

Golang course by Exadel

03 Nov 2022

Sergio Kovtunenko
Lead backend developer, Exadel
skovtunenko@exadel.com (mailto:skovtunenko@exadel.com)
https://github.com/skovtunenko (https://github.com/skovtunenko)
@realSKovtunenko (http://twitter.com/realSKovtunenko)