Methods and interfaces

This lesson covers methods and interfaces, the constructs that define objects and their behavior.

The Go Authors

https://golang.org

*** Methods**

Go does not have classes.

However, you can define methods on types.

A method is a function with a special _receiver_ argument.

The receiver appears in its own argument list between the `func` keyword and

the method name.

In this example, the `Abs` method has a receiver of type `Vertex` named `v`.

*** Methods are functions**

Remember: a method is just a function with a receiver argument.

Here's `Abs` written as a regular function with no change in functionality.

*** Methods continued**

You can declare a method on non-struct types, too.

In this example we see a numeric type `MyFloat` with an `Abs` method.

You can only declare a method with a receiver whose type is defined in the same package as the method.
You cannot declare a method with a receiver whose type is defined in another package (which includes the built-in types such as `int`).

*** Pointer receivers**

You can declare methods with pointer receivers.

This means the receiver type has the literal syntax `*T` for some type `T`.
(Also, `T` cannot itself be a pointer such as `*int`.)

For example, the `Scale` method here is defined on `*Vertex`.

Methods with pointer receivers can modify the value to which the receiver points (as `Scale` does here).
Since methods often need to modify their receiver, pointer receivers are more common than value receivers.

Try removing the `*` from the declaration of the `Scale` function on line 16 and observe how the program's behavior changes.

With a value receiver, the `Scale` method operates on a copy of the original `Vertex` value.
(This is the same behavior as for any other function argument.)
The `Scale` method must have a pointer receiver to change the `Vertex` value declared in the `main` function.

*** Pointers and functions**

Here we see the `Abs` and `Scale` methods rewritten as functions.

Again, try removing the `*` from line 16.

Can you see why the behavior changes?

What else did you need to change for the example to compile?

(If you're not sure, continue to the next page.)

*** Methods and pointer indirection**

Comparing the previous two programs, you might notice that

functions with a pointer argument must take a pointer:

```
 var v Vertex
 ScaleFunc(v, 5)  // Compile error!
 ScaleFunc(&v, 5) // OK
```

while methods with pointer receivers take either a value or a pointer as the

receiver when they are called:

```
 var v Vertex
 v.Scale(5)  // OK
 p := &v
 p.Scale(10) // OK
```

For the statement `v.Scale(5)`, even though `v` is a value and not a pointer,

the method with the pointer receiver is called automatically.

That is, as a convenience, Go interprets the statement `v.Scale(5)` as

`(&v).Scale(5)` since the `Scale` method has a pointer receiver.

*** Methods and pointer indirection (2)**

The equivalent thing happens in the reverse direction.

Functions that take a value argument must take a value of that specific type:

```
 var v Vertex

 fmt.Println(AbsFunc(v))  // OK

 fmt.Println(AbsFunc(&v)) // Compile error!
```

while methods with value receivers take either a value or a pointer as the

receiver when they are called:

```
 var v Vertex
 fmt.Println(v.Abs()) // OK

 p := &v

 fmt.Println(p.Abs()) // OK
```

In this case, the method call `p.Abs()` is interpreted as `(*p).Abs()`.

*** Choosing a value or pointer receiver**

There are two reasons to use a pointer receiver.

The first is so that the method can modify the value that its receiver points to.

The second is to avoid copying the value on each method call.
This can be more efficient if the receiver is a large struct, for example.

In this example, both `Scale` and `Abs` are methods with receiver type `*Vertex`,

even though the `Abs` method needn't modify its receiver.

In general, all methods on a given type should have either value or pointer

receivers, but not a mixture of both.

(We'll see why over the next few pages.)

*** Interfaces**

An _interface_type_ is defined as a set of method signatures.

A value of interface type can hold any value that implements those methods.

***Note:* There is an error in the example code on line 22.**

`Vertex` (the value type) doesn't implement `Abser` because

the `Abs` method is defined only on `*Vertex` (the pointer type).

*** Interfaces are implemented implicitly**

A type implements an interface by implementing its methods.

There is no explicit declaration of intent, no "implements" keyword.

Implicit interfaces decouple the definition of an interface from its

implementation, which could then appear in any package without prearrangement.

*** Interface values**

Under the hood, interface values can be thought of as a tuple of a value and a concrete type:

 (value, type)

An interface value holds a value of a specific underlying concrete type.

Calling a method on an interface value executes the method of the same name on its underlying type.

*** Interface values with nil underlying values**

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.)

Note that an interface value that holds a nil concrete value is itself non-nil.

*** Nil interface values**

A nil interface value holds neither value nor concrete type.

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.

*** The empty interface**

The interface type that specifies zero methods is known as the _empty_interface_:

```
 interface{}
```

An empty interface may hold values of any type.

(Every type implements at least zero methods.)

Empty interfaces are used by code that handles values of unknown type.

For example, `fmt.Print` takes any number of arguments of type `interface{}`.

*** Type assertions**

A _type_assertion_ provides access to an interface value's underlying concrete value.

```
 t := i.(T)
```

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.

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.

```
 t, ok := i.(T)
```

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.

Note the similarity between this syntax and that of reading from a map.

*** Type switches**

A _type_switch_ is a construct that permits several type assertions in series.

A type switch is like a regular switch statement, but the cases in a type switch specify types (not values), and those values are compared against the type of the value held by the given interface value.

```
 switch v := i.(type) {
 case T:
  // here v has type T
 case S:
  // here v has type S
 default:
  // no match; here v has the same type as i
 }
```

The declaration in a type switch has the same syntax as a type assertion `i.(T)`,

but the specific type `T` is replaced with the keyword `type`.

This switch statement tests whether the interface value `i`

holds a value of type `T` or `S`.

In each of the `T` and `S` cases, the variable `v` will be of type

`T` or `S` respectively and hold the value held by `i`.

In the default case (where there is no match), the variable `v` is

of the same interface type and value as `i`.

*** Stringers**

One of the most ubiquitous interfaces is [[/pkg/fmt/#Stringer][`Stringer`]] defined by the [[/pkg/fmt/][`fmt`]] package.

```
 type Stringer interface {

  String() string

 }
```

A `Stringer` is a type that can describe itself as a string. The `fmt` package

(and many others) look for this interface to print values.

*** Exercise: Stringers**

Make the `IPAddr` type implement `fmt.Stringer` to print the address as

a dotted quad.

For instance, `IPAddr{1,`2,`3,`4}` should print as `"1.2.3.4"`.

*** Errors**

Go programs express error state with `error` values.

The `error` type is a built-in interface similar to `fmt.Stringer`:

```
 type error interface {
  Error() string
 }
```

(As with `fmt.Stringer`, the `fmt` package looks for the `error` interface when

printing values.)

Functions often return an `error` value, and calling code should handle errors

by testing whether the error equals `nil`.

```
 i, err := strconv.Atoi("42")

 if err != nil {

  fmt.Printf("couldn't convert number: %v\n", err)

  return

 }

 fmt.Println("Converted integer:", i)
```

A nil `error` denotes success; a non-nil `error` denotes failure.

*** Exercise: Errors**

Copy your `Sqrt` function from the [[/tour/flowcontrol/8][earlier exercise]] and modify it to return an `error` value.

`Sqrt` should return a non-nil error value when given a negative number, as it doesn't support complex numbers.

Create a new type

```
 type ErrNegativeSqrt float64
```

and make it an `error` by giving it a

```
 func (e ErrNegativeSqrt) Error() string
```

method such that `ErrNegativeSqrt(-2).Error()` returns
`"cannot`Sqrt`negative`number:`-2"`.

***Note:* A call to `fmt.Sprint(e)` inside the `Error` method will send the program into an infinite loop. You can avoid this by converting `e` first: `fmt.Sprint(float64(e))`. Why?**

Change your `Sqrt` function to return an `ErrNegativeSqrt` value when given a negative number.

*** Readers**

The `io` package specifies the `io.Reader` interface,

which represents the read end of a stream of data.

The Go standard library contains
[[https://cs.opensource.google/search?q=Read%5C(%5Cw%2B%5Cs%5C%5B%5C%5Dbyte
implementations]] of this interface, including files, network connections, compressors, ciphers, and others.
The `io.Reader` interface has a `Read` method:

```
 func (T) Read(b []byte) (n int, err error)
```

`Read` populates the given byte slice with data and returns the number of bytes

populated and an error value. It returns an `io.EOF` error when the stream

ends.

The example code creates a

[[/pkg/strings/#Reader][`strings.Reader`]]

and consumes its output 8 bytes at a time.

*** Exercise: Readers**

Implement a `Reader` type that emits an infinite stream of the ASCII character

`'A'`.

*** Exercise: rot13Reader**

A common pattern is an [[/pkg/io/#Reader][io.Reader]] that wraps another
`io.Reader`, modifying the stream in some way.

For example, the [[/pkg/compress/gzip/#NewReader][gzip.NewReader]] function
takes an `io.Reader` (a stream of compressed data) and returns a `*gzip.Reader`
that also implements `io.Reader` (a stream of the decompressed data).

Implement a `rot13Reader` that implements `io.Reader` and reads from an `io.Reader`, modifying the stream by applying the [[https://en.wikipedia.org/wiki/ROT13][rot13]] substitution cipher to all alphabetical characters.
The `rot13Reader` type is provided for you.

Make it an `io.Reader` by implementing its `Read` method.

*** Images**

[[/pkg/image/#Image][Package image]] defines the `Image` interface:

```
 package image

 type Image interface {
  ColorModel() color.Model
  Bounds() Rectangle
  At(x, y int) color.Color
 }
```

***Note*: the `Rectangle` return value of the `Bounds` method is actually an**

[[/pkg/image/#Rectangle][`image.Rectangle`]], as the

declaration is inside package `image`.

(See [[/pkg/image/#Image][the documentation]] for all the details.)

The `color.Color` and `color.Model` types are also interfaces, but we'll ignore that by using the predefined implementations `color.RGBA` and `color.RGBAModel`. These interfaces and types are specified by the [[/pkg/image/color/][image/color package]]

*** Exercise: Images**

Remember the [[/tour/moretypes/18][picture generator]] you wrote earlier? Let's write another one, but this time it will return an implementation of `image.Image` instead of a slice of data.

Define your own `Image` type, implement [[/pkg/image/#Image][the necessary methods]], and call `pic.ShowImage`.

`Bounds` should return a `image.Rectangle`, like `image.Rect(0,`0,`w,`h)`.

`ColorModel` should return `color.RGBAModel`.

`At` should return a color; the value `v` in the last picture generator corresponds to `color.RGBA{v,`v,`255,`255}` in this one.

*** Congratulations!**

You finished this lesson!

You can go back to the list of [[/tour/list][modules]] to find what to learn next, or continue with the [[javascript:click('.next-page')][next lesson]].