

The Go Programming Language

Part 2

Rob Pike
r@google.com
October, 2009



Today's Outline

Exercise

any questions?

Composite types

structures, arrays, slices, maps

Methods

they're not just for structs any more

Interfaces



Exercise

Any questions?

One nugget:

`n0, n1 = n0+n1, n0`

or for a general binary operator

`n0, n1 = op(n0, n1), n0`



Composite types

A hand-drawn graphic on a black background. It features the word "GO" in a thick, white, chalk-like font. To the left of the "G", there are two parallel white lines that taper to the left, suggesting motion or speed.

Arrays

Arrays are quite different from C arrays; more like Pascal arrays. (Slices, the next topic, act a little more like C arrays.)

```
var ar [3]int
```

declares `ar` to be an array of 3 integers, initially all set to 0.

Size is part of the type.

Built in function `len()` reports size:

```
len(ar) == 3
```



Arrays are values

Arrays are values, not implicit pointers as in C. You can take an array's address (for instance, to pass it efficiently to a function):

```
func f(a [3]int) { fmt.Println(a) }
func fp(a *[3]int) { fmt.Println(a) }

func main() {
    var ar [3] int;
    f(ar);      // passes a copy of ar
    fp(&ar);   // passes a pointer to ar
}
```

Output (Print[ln] knows about arrays!):

```
[0 0 0]
&[0 0 0]
```



Array literals

All the composite types have similar syntax for creating values. For arrays the literal looks like this:

Array of 3 integers:

```
[3]int { 1, 2, 3 }
```

Array of 10 integers, first three not zero:

```
[10]int { 1, 2, 3 }
```

Don't want to count? Use ... for the length:

```
[...]int { 1, 2, 3 }
```

Don't want to initialize them all? Use key:value pairs:

```
[10]int { 2:1, 3:1, 5:1, 7:1 }
```



Pointers to array literals

You can take the address of an array literal to get a pointer to a newly created instance:

```
func fp(a *[3]int) { fmt.Println(a) }

func main() {
    for i := 0; i < 3; i++ {
        fp(&[3]int{i, i*i, i*i*i})
    }
}
```

Output:

```
&[0 0 0]
&[1 1 1]
&[2 4 8]
```



Slices

A slice is a **reference** to a section of an array.
Slices are used much more often than plain arrays.

A slice is very cheap. (More about this soon.)

A slice type looks like an array type without a size:

```
var a []int
```

`len()` returns the number of elements.

Create a slice by "slicing" an array or slice:

```
a = ar[7:9];
```

Valid indexes of `a` will be `0`, `1` and `len(a)==2`.

Can also just assign an array pointer to a slice:

```
a = &ar; // same as a = ar[0:len(ar)]
```



A slice references an array

Conceptually:

```
type Slice struct {  
    base *elem_type;  
    len  int;  
    cap  int;  
}
```

Array:

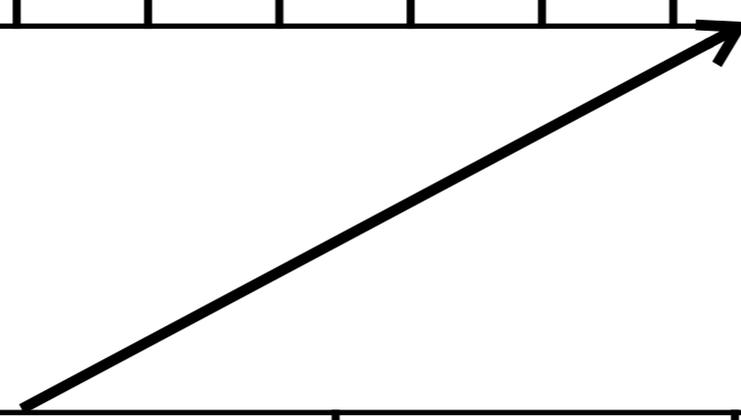
ar:

7	1	5	4	3	8	7	2	11	5	3
---	---	---	---	---	---	---	---	----	---	---

Slice:

a=ar[7:9]:

base=&ar[7]	len=2	cap=4
-------------	-------	-------



Making slices

Slice literals look like array literals without a size:

```
var slice = []int{ 1,2,3,4,5 }
```

What this does is create an array of length 5 and then create a slice to refer to it.

We can also allocate a slice (and underlying array) with the built in function `make()`:

```
var s100 = make([]int, 100) // slice: 100 ints
```

Why `make` not `new`? Because we need to make a slice, not just allocate the memory. Note `make([]int)` returns `[]int` while `new([]int)` returns `*[]int`.

`make` is used for slices, maps, and channels.



Slice capacity

A slice refers to an underlying array, so there may be elements off the end of the slice that are present in the array.

The built in function `cap()` (capacity) reports how long the slice can grow. After

```
var ar = [10]int{0,1,2,3,4,5,6,7,8,9}
var a = &ar[5:7] // reference to subarray {5,6}
```

`len(a)` is 2 and `cap(a)` is 5. We can "reslice":

```
a = a[0:4] // ref of subarray {5,6,7,8}
```

`len(a)` is now 4 but `cap(a)` is still 5.



Resizing a slice

Slices can be used like growable arrays. Allocate one using `make()` with two numbers – length and capacity – and reslice as it grows:

```
var s1 = make([]int, 0, 100) // len 0, cap 100

func appendToSlice(i int, s1 []int) []int {
    if len(s1) == cap(s1) { error(...) }
    n := len(s1);
    s1 = s1[0:n+1]; // extend length by 1
    s1[n] = i;
    return s1
}
```

Thus `s1`'s length is always the number of elements, but it grows as needed.

This style is cheap and idiomatic in Go.



Slices are cheap

Feel free to allocate and resize slices as required. They are cheap to pass around; no need to allocate.

Remember they are references, so underlying storage can be modified.

For instance, I/O uses slices, not counts:

```
func Read(fd int, b []byte) int
var buffer [100]byte
for i := 0; i < 100; i++ {
    // Fill buffer one byte at a time.
    Read(fd, buffer[i:i+1]) // no allocation here
}
```

Split a buffer:

```
header, data := buf[0:n], buf[n:len(buf)]
```

Strings can be sliced too, with similar efficiency.



Maps

Maps are another reference type. They are declared like this:

```
var m map[string] float
```

This declares a map indexed with key type `string` and value type `float`. It is analogous to the C++ type `*map<string, float>` (note the `*`).

On a map, `len()` returns the number of keys.



Map creation

As with a slice, a map variable refers to nothing; you must put something in it before it can be used.

Three ways.

1) Literal: list of colon-separated key:value pairs

```
m = map[string]float { "1":1, "pi":3.1415 }
```

2) Creation

```
m = make(map[string]float) // make not new
```

3) Assignment

```
var m1 map[string]float  
m1 = m // m1 and m now refer to same map
```



Indexing a map

```
m = map[string]float { "1":1, "pi":3.1415 }
```

Access an element as a value:

```
one    := m["1"]  
crash := m["not present"] // error
```

Set an element (setting twice updates value for key)

```
m["2"] = 2;  
m["2"] = 3; // mess with their heads
```



Testing existence

To see if a key is present in the map, we use a multi-value assignment, the "comma ok" form:

```
m = map[string]float { "1":1, "pi":3.1415 };  
  
var value float;  
var present bool;  
  
value, present = m[x];
```

or idiomatically

```
v, ok := m[x]; // hence, the "comma ok" form
```

If `x` is present in the map, sets the boolean to `true` and the value to the entry for the key. If not, sets the boolean to `false` and the value to the zero for its type.



Deleting

Deleting an entry in the map is a multi-variate assignment **to** the map entry:

```
m = map[string]float { "1":1.0, "pi":3.1415 };  
  
var present bool;  
var value float;  
var x string = f();  
  
m[x] = v, present;
```

If `present` is `true`, assigns `v` to the map; if `present` is `false`, deletes the entry for key `x`. So to delete an entry,

```
m[x] = 0, false // deletes entry for x
```



For and range

The `for` loop has a special syntax for iterating over arrays, slices, maps (and more, as we'll see tomorrow).

```
m := map[string]float { "1":1.0, "pi":3.1415 };  
for key, value := range m {  
    fmt.Printf("key %s, value %g\n", key, value)  
}
```

With only one variable in the `range`, get the key:

```
for key = range m {  
    fmt.Printf("key %s\n", key)  
}
```

Variables can be assigned or `:=` declared.

For arrays and slices, get index and value.



Range over string

A `for` using `range` on a string loops over Unicode code points, not bytes. (Use `[]byte` for bytes). The string is assumed to contain UTF-8.

The loop

```
s := "[\u00ff\u754c]";  
for i, c := range s {  
    fmt.Printf("%d:%c ", i, c)  
}
```

Prints 0:[1:ÿ 3:界 6:]

If erroneous UTF-8 is encountered, the character is set to U+FFFD and the index advances by one byte.



Structs

Structs should feel very familiar: simple declarations of data fields.

```
var p struct { x, y float }
```

More usual:

```
type Point struct { x, y float }  
var p Point
```

Structs allow the programmer to define the layout of memory.



Structs are values

Structs are values and `new(StructType)` returns a pointer to a value.

```
type Point struct { x, y float };  
var p Point;  
p.x = 7;  
p.y = 23.4;  
var pp *Point = new(Point);  
*pp = p;  
pp.x = Pi; // sugar for (*pp).x
```

There is no `->` notation for structure pointers. Go provides the indirection for you.



Making structs

Structs are values so you can make a zeroed one just by declaring it.

You can also allocate one with `new()`.

```
var p Point;           // zeroed value
pp := new(Point);     // idiomatic allocation
```

Struct literals have the expected syntax.

```
p = Point{ 7.2, 8.4 }
p = Point{ y:8.4, x:7.2 }
pp := &Point{ 23.4, -1 } // idiomatic
```

As with arrays, taking the address of a struct literal gives the address of a newly created value.

This last example is a constructor for `*Point`.



Exporting types and fields

The fields (and methods, coming up soon) of a `struct` must start with an Uppercase letter to be visible outside the package.

Private type and fields:

```
type point struct { x, y float }
```

Exported type and fields:

```
type Point struct { X, Y float }
```

Exported type with mix of fields:

```
type Point struct {  
    X, Y float;    // exported  
    name string;  // not exported  
}
```

You may even have a private type with exported fields. Exercise: why is that useful?



Anonymous fields

Inside a struct, you can declare fields, such as another struct, without giving a name for the field. These are called anonymous fields and they act as if the inner struct is simply inserted or "embedded" into the outer.

This simple mechanism provides a way to derive some or all of your implementation from another type or types.

An example follows.



An anonymous struct field

```
type A struct {  
    ax, ay int  
}  
type B struct {  
    A;  
    bx, by float;  
}
```

B acts as if it has four fields, ax, ay, bx, and by. It's almost as if B is { ax, ay int; bx, by float }.

However, literals for B must be filled out in detail:

```
b := B{ A{ 1, 2 }, 3.0, 4.0 };  
fmt.Println(b.ax, b.ay, b.bx, b.by);
```

Prints 1 2 3 4



Anonymous fields have type as name

But it's richer than simple interpolation of the fields: B also has a field A. The anonymous field looks like a field whose name is its type.

```
b := B{ A{ 1, 2 }, 3.0, 4.0 };  
fmt.Println(b.A);
```

Prints {1 2}. If A came from another package, the field would still be called A:

```
import "pkg"  
type C struct { pkg.A }  
...  
c := C { pkg.A{ 1, 2 } };  
fmt.Println(c.A); // not c.pkg.A
```



Anonymous fields of any type

Any named type, or pointer to one, may be used in an anonymous field and it may appear at any location in the struct.

```
type C struct {  
    x float;  
    int;  
    string;  
}
```

```
c := C{ 3.5, 7, "hello" };  
fmt.Println(c.x, c.int, c.string)
```

Prints 3.5 7 hello



Conflicts and hiding

If there are two fields with the same name (possibly a type-derived name), these rules apply:

1) An outer name hides an inner name.

This provides a way to override a field/method.

2) If the same name appears twice at the same level, it is an error if the name is used by the program. (If it's not used, it doesn't matter.)

No rules to resolve the ambiguity; it must be fixed.



Conflict examples

```
type A struct { a int }  
type B struct { a, b int }
```

```
type C struct { A; B }  
var c C;
```

Using `c.a` is an error: is it `c.A.a` or `c.B.a`?

```
type D struct { B; b float }  
var d D;
```

Using `d.b` is OK: it's the `float`, not `d.B.b`
Can get at the inner `b` by `D.B.b`.



Methods



Methods on structs

Go has no classes, but you can attach methods to any type. Yes, (almost) any type. The methods are declared, separate from the type declaration, as functions with an explicit receiver. The obvious struct case:

```
type Point struct { x, y float }  
  
// A method on *Point  
func (p *Point) Abs() float {  
    return math.Sqrt(p.x*p.x + p.y*p.y)  
}
```

Note: explicit receiver (no `this`), in this case of type `*Point`, used within the method.



Methods on struct values

A method does not require a pointer as a receiver.

```
type Point3 struct { x, y, z float }

// A method on Point3
func (p Point3) Abs() float {
    return math.Sqrt(p.x*p.x + p.y*p.y + p.z*p.z)
}
```

This is a bit expensive, because the `Point3` will always be passed to the method by value, but it is valid Go.



Invoking a method

Just as you expect.

```
p := &Point{ 3, 4 };  
fmt.Print(p.Abs()); // will print 5
```

A non-struct example:

```
type IntVector []int  
func (v IntVector) Sum() (s int) {  
    for i, x := range v {  
        s += x  
    }  
    return  
}  
  
fmt.Println(IntVector{1, 2, 3}.Sum())
```



Ground rules for methods

Methods are attached to a named type, say `Foo`, and are statically bound.

The type of a receiver in a method can be either `*Foo` or `Foo`. You can have some `Foo` methods and some `*Foo` methods.

`Foo` itself cannot be a pointer type, although the methods can have receiver type `*Foo`.

The type `Foo` must be defined in the same package as all its methods.



Pointers and values

Go automatically indirections/dereferences values for you when invoking methods.

For instance, even though a method has receiver type `*Point` you can invoke it on an addressable value of type `Point`.

```
p1 := Point{ 3, 4 };  
fmt.Print(p1.Abs()); // sugar for (&p1).Abs()
```

Similarly, if methods are on `Point3` you can use a value of type `*Point3`:

```
p3 := &Point3{ 3, 4, 5 };  
fmt.Print(p3.Abs()); // sugar for (*p3).Abs()
```



Methods on anonymous fields

Naturally, when an anonymous field is embedded in a struct, the methods of that type are embedded as well – in effect, it inherits the methods.

This mechanism offers a simple way to emulate some of the effects of subclassing and inheritance.



Anonymous field example

```
type Point struct { x, y float }  
func (p *Point) Abs() float { ... }
```

```
type NamedPoint struct {  
    Point;  
    name string;  
}
```

```
n := &NamedPoint{Point{3, 4}, "Pythagoras"};  
fmt.Println(n.Abs()); // prints 5
```



Overriding a method

Overriding works just as with fields.

```
type NamedPoint struct {  
    Point;  
    name string;  
}  
func (n *NamedPoint) Abs() float {  
    return n.Point.Abs() * 100.  
}
```

```
n := &NamedPoint{Point{3, 4}, "Pythagoras"};  
fmt.Println(n.Abs()); // prints 500
```

Of course you can have multiple anonymous fields with various types – a simple version of multiple inheritance. The conflict resolution rules keep things simple, though.



Another example

A more compelling use of an anonymous field.

```
type Mutex struct { ... }
func (m *Mutex) Lock() { ... }

type Buffer struct {
    data [100]byte;
    Mutex; // need not be first in Buffer
}
var buf = new(Buffer);
buf.Lock(); // == buf.Mutex.Lock()
```

Note that `Lock`'s receiver is (the address of) the `Mutex` field, not the surrounding structure. (Contrast to subclassing or Lisp mix-ins.)



Other types

Methods are not just for structs. They can be defined for any (non-pointer) type.

The type must be defined in your package though. You can't write a method for `int` but you can declare a new `int` type and give it methods.

```
type Day int
```

```
var dayName = []string {  
    "Monday", "Tuesday", "Wednesday", ...  
}
```

```
func (day Day) String() string {  
    return dayName[day]  
}
```



Other types

Now we have an enum-like type that knows how to print itself.

```
const (  
    Monday Day = iota;  
    Tuesday;  
    Wednesday;  
    ...  
)  
  
var day = Tuesday;  
fmt.Print(day.String()); // prints Tuesday
```



Print() understands String()

By techniques to be divulged soon, `fmt.Println` can identify values that implement the method `String()` as we defined for type `Day`. Such values are automatically formatted by invoking the method. Thus:

```
fmt.Println(0, Monday, 1, Tuesday)
```

prints 0 Monday 1 Tuesday.

`Println` can tell a plain 0 from a 0 of type `Day`.

So define a `String()` method for your types and they will print nicely with no more work.



Visibility of fields and methods

Go is very different from C++ in the area of visibility.
The rules:

- 1) Go has package scope (C++ has file scope).
- 2) Spelling determines exported/local (pub/priv).
- 3) Structs in the same package have full access to one another's fields and methods.
- 4) Local type can export its fields and methods.
- 5) No true subclassing, so no notion of protected.

These simple rules seem to work well in practice.



Interfaces

A hand-drawn graphic on a black background. It features the word "GO" in a white, chalk-like, cursive font. To the left of the "G", two parallel white lines with arrowheads point towards the letter, suggesting motion or direction.

Watch closely

We are about to look at Go's most unusual aspect:
the interface.

Please leave your preconceptions at the door.



Introduction

So far, all the types we have examined have been concrete: they implement something.

There is one more type to consider: the interface type. It is completely abstract; it implements nothing. Instead, it specifies a set of properties an implementation must provide.

Interface as a concept is very close to that of Java, and Java has an interface type, but the "interface value" of Go is novel.



Definition of an interface

The word "interface" is a bit overloaded in Go: there is the concept of an interface, and there is an interface type, and then there are values of that type. First, the concept.

Definition:

An interface is a set of methods.

To turn it around, the methods implemented by a concrete type such as a `struct` form the interface of that type.



Example

We saw this simple type before:

```
type Point struct { x, y float }  
func (p *Point) Abs() float { ... }
```

The interface of type `Point` is just one method:

```
Abs() float
```

It's not

```
func (p *Point) Abs() float
```

because the interface abstracts away the receiver.

We embedded `Point` in a new type, `NamedPoint`;
`NamedPoint` has the same interface.



The interface type

An interface type is a specification of an interface, a set of methods implemented by some other types. Here's a simple one, with only one method:

```
type AbsInterface interface {  
    Abs() float // receiver is implied  
}
```

This is a definition of the interface implemented by `Point`, or in our terminology,

```
Point implements AbsInterface
```

Also,

```
NamedPoint and Point3 implement AbsInterface
```

Methods are listed inside the type, not outside. 

An example

```
type MyFloat float

func (f MyFloat) Abs() float {
    if f < 0 { return -f }
    return f
}
```

MyFloat implements AbsInterface even though float does not.



Many to many

An interface may be implemented by an arbitrary number of types. `AbsInterface` is implemented by any type that has a method `Abs() float`, regardless of what other methods that type may have.

A type may implement an arbitrary number of interfaces. `Point` implements at least these two:

```
type AbsInterface interface { Abs() float }  
type EmptyInterface interface { }
```

And perhaps more, depending on its methods.

Every type implements `EmptyInterface`. That will come in handy.



Interface value

Once a variable is declared with interface type, it may store any value that implements that interface.

```
var ai AbsInterface;

pp := new(Point);
ai = pp; // OK: *Point has Abs()
ai = 7.; // compile-time err: float has no Abs()
ai = MyFloat(-7.); // OK: MyFloat has Abs()

ai = &Point{ 3, 4 };
fmt.Printf(ai.Abs()); // method call
```

Prints 5.

Note: ai is not a pointer.



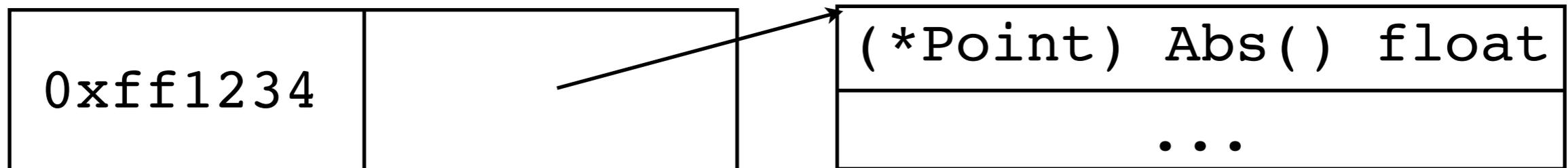
In memory

ai is not a pointer! It's a multiword data structure.

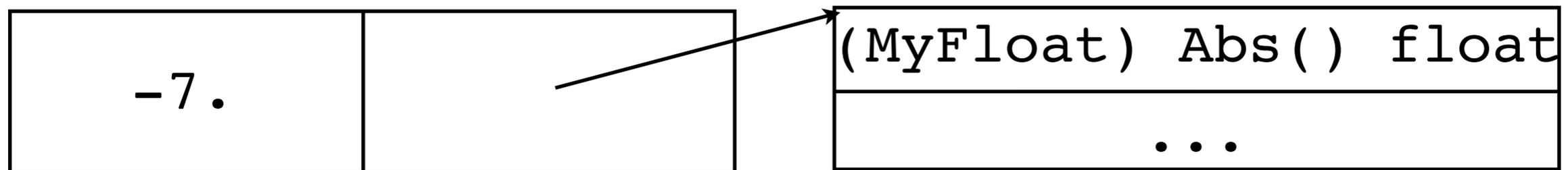


At different times it has different value and type:

ai = &Point{3,4} (==(*Point)(0xff1234)):



ai = MyFloat (-7.):



Three important facts

1) Interfaces define sets of methods. They are pure and abstract: no implementation, no data fields. Go has a clear separation between interface and implementation.

2) Interface values are just that: values. They contain any concrete value that implements all the methods defined in the interface. **That value may or may not be a pointer.**

3) Types implement interfaces just by having methods. They do not have to declare that they do so. For instance, every type implements `EmptyInterface`.



Example: io.Writer

Here is the actual signature of `fmt.Fprintf`:

```
func Fprintf(w io.Writer, format string, a ...)
           (n int, error os.Error)
```

It doesn't write to a file, it writes to something of type `io.Writer`, that is, `Writer` defined in the `io` package:

```
type Writer interface {
    Write(p []byte) (n int, err os.Error)
}
```

`Fprintf` can therefore be applied to any type that has a canonical `Write` method, including files, pipes, network connections, and...



Buffered I/O

... a write buffer. This is from the `bufio` package:

```
type Writer struct { ... }
```

`bufio.Writer` implements the canonical `Write` method.

```
func (b *Writer) Write(p []byte)
                        (nn int, err os.Error)
```

It also has a factory: give it an `io.Writer`, it will return a buffered `io.Writer` in the form of a `bufio.Writer`:

```
func NewWriter(wr io.Writer)
                (b *Writer, err os.Error)
```

And of course, `os.File` implements `Write` too.



Putting it together

```
import (
    "bufio"; "fmt"; "os"
)
func main() {
    // unbuffered
    fmt.Fprintf(os.Stdout, "%s, ", "hello");
    // buffered: os.Stdout implements io.Writer
    buf, err := bufio.NewWriter(os.Stdout);
    // and now so does buf.
    fmt.Fprintf(buf, "%s\n", "world!");
}
```

Buffering works for anything that `writes`.

Feels almost like Unix pipes, doesn't it? The composability is powerful; see `crypto` packages. 

Other public interfaces in io

The `io` package has:

`Reader`

`Writer`

`ReadWriter`

`ReadWriteCloser`

These are stylized interfaces but obvious: they capture the functionality of anything implementing the functions listed in their names.

This is why we can have a buffered I/O package with an implementation separate from the I/O itself: it both accepts and provides interface values.



Comparison

In C++ terms, an interface type is like a pure abstract class, specifying the methods but implementing none of them.

In Java terms, an interface type is much like a Java interface.

However, in Go there is a major difference:

A type does not need to declare the interfaces it implements or to inherit from an interface type; if it has the methods, it implements the interface.

Some other differences will become apparent.



Anonymous fields work too

```
type LockedBufferedWriter struct {
    Mutex; // has Lock and Unlock methods
    bufio.Writer; // has Write method
}

func (l *LockedBufferedWriter) Write(p []byte)
    (nn int, err os.Error) {
    l.Lock();
    defer l.Unlock();
    return l.Writer.Write(p); // inner Write()
}
```

LockedBufferedWriter implements `io.Writer`, but also, through the anonymous `Mutex`,

```
type Locker interface { Lock(); Unlock() }
```



Example: HTTP service

```
type Handler interface {  
    ServeHTTP(*Conn, *Request)  
}
```

This is the interface defined by the HTTP server package. To serve HTTP, define a type that implements this interface and connect it to the server (details omitted).

```
type Counter struct {  
    n int // or could just say type Counter int  
}  
  
func (ctr *Counter) ServeHTTP(c *http.Conn,  
                               req *http.Request) {  
    fmt.Fprintf(c, "counter = %d\n", ctr.n);  
    ctr.n++;  
}
```



A function (type) that serves HTTP

```
func notFound(c *Conn, req *Request) {
    c.SetHeader("Content-Type", "text/plain;"
                "charset=utf-8");
    c.WriteHeader(StatusNotFound);
    c.WriteString("404 page not found\n");
}
```

Now we define a type to implement ServeHTTP:

```
type HandlerFunc func(*Conn, *Request)
func (f HandlerFunc) ServeHTTP(c *Conn, req
                                *Request) {
    f(c, req) // the receiver's a func; call it
}
```

Convert function to attach method, implement the interface:

```
var Handle404 = HandlerFunc(notFound);
```



Containers & the empty interface

Sketch of the implementation of vectors:

```
type Element interface {}

// Vector is the container itself.
type Vector struct {
    a []Element
}

// At() returns the i'th element.
func (p *Vector) At(i int) Element {
    return p.a[i]
}
```

Vectors can contain anything because any type implements the empty interface. (In fact every element could be of different type.)



Type assertions

Once you put something into a `vector`, it's stored as an interface value. Need to "unbox" it to get the original back: use a "type assertion". Syntax:

```
interface_value.(type_to_extract)
```

Will trap if type is wrong – but see next slide.

```
var v vector.Vector;
v.Set(0, 1234.); // stored as interface val
i := v.At(0);   // retrieved as interface{}
if i != 1234. {} // compile-time err
if i.(float) != 1234. {} // OK
if i.(int) != 1234 {} // run-time err
if i.(MyFloat) != 1234. {} // err: not MyFloat
```

Type assertions always execute at run time. Compiler rejects assertions guaranteed to fail.



Interface to interface conversion

So far we've only moved regular values into and out of interface values, but interface values that contain the appropriate methods can also be converted.

In effect, it's the same as unboxing the interface value to extract the underlying concrete value, then boxing it again for the new interface type.

The conversion's success depends on the underlying value, not the original interface type.



Interface to interface example

Given

```
var ai AbsInterface
type SqrInterface interface { Sqr() float }
var si SqrInterface
pp := new(Point) // say *Point has Abs, Sqr
var empty interface{}
```

These are all OK

```
empty = pp; // everything satisfies empty
ai = empty.(AbsInterface); // underlying
// value implements Abs()
// (runtime failure otherwise)
si = ai.(SqrInterface); // *Point has Sqr()
// even though AbsInterface doesn't
empty = si; // *Point implements empty set
// Note: statically checkable
// so type assertion not necessary.
```



Testing with type assertions

Can use "comma ok" type assertions to test a value for type.

```
elem := vector.At(0);
if i, ok := elem.(int); ok {
    fmt.Printf("int: %d\n", i)
}
else if f, ok := elem.(float); ok {
    fmt.Printf("float: %g\n", f)
}
else {
    fmt.Print("unknown type\n")
}
```



Testing with a type switch

Special syntax:

```
switch v := elem.(type) { // literal "type"  
case int:  
    fmt.Printf("is int: %d\n", v);  
case float:  
    fmt.Printf("is float: %g\n", v);  
default:  
    fmt.Print("unknown type\n");  
}
```



Does v implement m()?

Going one step further, can test whether a value implements a method.

```
type Stringer interface { String() string }

if sv, ok := v.(Stringer); ok {
    fmt.Printf("implements String(): %s\n",
              sv.String()); // note: sv not v
}
```

This is how `Print` etc. check if type can print itself.



Reflection and ...

There is a reflection package ("reflect") that builds on these ideas to let you examine values to discover their type. Too intricate to describe here but `Printf` etc. use it to unpack the ... arguments.

```
func Printf(format string, args ...)
    (n int, err os.Error)
```

The ... argument inside `Printf` (or anywhere else) has type `interface{}`, and `Printf` uses the reflection package to unpack it and discover the argument list.



Reflection and Print

As a result, `Printf` and its friends know the actual types of their arguments. Because they know if the argument is unsigned or long, there is no `%u` or `%ld`, only `%d`.

This is also how `Print` and `Println` can print the arguments nicely without a format string.

There is also a `%v` ("value") format that gives default nice output from `Printf` for values of any type.

```
fmt.Printf("%v %v %v %v", -1, "hello",  
           []int{1,2,3}, uint64(456))
```

Prints `-1 hello [1,2,3] 456`.

In fact, `%v` is identical to the formatting done by `Print` and `Println`.



Exercise



Exercise: Day 2

Look at the `http` package.

Write an HTTP server to present pages in the file system, but transformed somehow, perhaps `rot13`, perhaps something more imaginative. Can you make the transformation substitutable? Can you work in your Fibonacci program somehow?



Next lesson

Concurrency and communication



The Go Programming Language

Part 2

Rob Pike
r@google.com
October, 2009

