Methods and Interfaces

Objectives

- Explore:
  - Methods
  - Receivers
  - Method binding
  - Encapsulation
  - Interfaces
  - Type Assertions

Methods

- There is no universal definition of object-oriented programming
- In Go, an object is a variable that has methods
- A method is a function associated with a particular type
  - An object-oriented program is one that uses methods to express the properties and operations of each data structure so that clients need not access the object's representation directly
  - This is one of the key features of object orientation: Encapsulation
- A method is declared much like an ordinary function but with an extra parameter before the function name
  - This parameter attaches the function to the type
  - The type parameter name is called the receiver and is like the "this" or "self" pointer in some other OO languages
    - receiver names are commonly the first letter of the type name
- Methods are invoked using . notation on the desired object: bike.mm()
  - Such an expression is called a selector in Go
- Each type has its own namespace allowing multiple types to have the same method names
  - Methods can also use names already taken at the package level
  - This allows method names to be compact
- Methods may be declared on any named type defined in the same package, so long as its underlying type is neither a pointer nor an interface

```go
package main

import "fmt"

type Moto struct {
    Make  string
    Model string
}

func (m Moto) mm() string {
    return m.Make + " " + m.Model
}

type digit int

func (i digit) prt() string {
    return fmt.Sprintf("int: %d", i)
}

func main() {
    bike := Moto{"Honda", "VFR750"}
    fmt.Println(bike.mm())

    var x digit = 7
    fmt.Println(x.prt())
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
2017/04/10 22:05:30 server.go:73: Using API v1
2017/04/10 22:05:30 debugger.go:68: launching pro
API server listening at: 127.0.0.1:2345
2017/04/10 22:05:31 debugger.go:414: continuing
Honda VFR750
int: 7
```

Pointer Receivers

- Calling a function makes a copy of each argument value
- If a function needs to update a variable or if an argument is large, passing the address of the variable using a pointer is desirable
- Methods that need to update the receiver variable need a pointer receiver
  - Even when pointer receivers are defined methods can be called with the name directly
    - bike.ModelSuffix()
      - As opposed to:
    - (&bike).ModelSuffix()
- Nil can be a legal receiver value for types with a nil zero value

```go
package main

import "fmt"

// Moto is for motorcycle
type Moto struct {
	Make  string
	Model string
}

func (m Moto) mm() string {
	return m.Make + " " + m.Model
}

func (m *Moto) ModelSuffix(suf string) {
	m.Model += suf
}

func main() {
	bike := Moto{"Honda", "VFR750"}
	fmt.Println(bike.mm())
	bike.ModelSuffix("F")
	fmt.Println(bike.mm())
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
2017/04/10 22:16:56 server.go:73: Using API v1
2017/04/10 22:16:56 debugger.go:68: launching proc
API server listening at: 127.0.0.1:2345
2017/04/10 22:16:56 debugger.go:414: continuing
Honda VFR750
Honda VFR750F
```

Binding Methods

- It's possible to separate method/receiver binding from invocation
- The selector p.mm yields a function with the method mm bound to variable p
  - This function can then be invoked without a receiver value
    - x := p.mm
    - x()
- The selector Moto.mm yields a function invoking the method mm with an additional parameter prepended to accept the receiver
  - x := Moto.mm
  - x(bike)
- Binding methods and objects is shockingly simple compared to the process in other languages (C++98 anyone?)

```go
package main

import "fmt"

// Moto is for motorcycle
type Moto struct {
    Make  string
    Model string
}

func (m Moto) mm() string {
    return m.Make + " " + m.Model
}

func (m *Moto) ModelSuffix(suf string) {
    m.Model += suf
}

func main() {
    bike := Moto{"Honda", "VFR750"}

    bmm := Moto.mm
    fmt.Println(bmm(bike))

    bms := bike.ModelSuffix
    bms("F")
    fmt.Println(bike.mm())
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
2017/04/10 22:37:49 server.go:73: Using API v1
2017/04/10 22:37:49 debugger.go:68: launching proce
API server listening at: 127.0.0.1:2345
2017/04/10 22:37:50 debugger.go:414: continuing
Honda VFR750
Honda VFR750F
```

Encapsulation in Go

- In Go the unit of encapsulation is the package
  - NOT the type
  - The fields of a struct type are visible to all code within the same package
- Go has only one mechanism to control the visibility of names:
  - Identifier Capitalization
    - Capitalized names are exported from the package
    - Uncapitalized names are not exported from the package
  - This rule also applies to the fields of a struct or the methods of a type
- To encapsulate an object we must make it a struct and give the attribute(s) lowercase names
  - It is common to have structs with a single field for this reason
- Encapsulation provides three benefits
  - one need inspect fewer statements to understand the possible values of variables
  - hiding implementation details prevents clients from depending on things that might change
  - clients are prevented from setting an object's variables arbitrarily

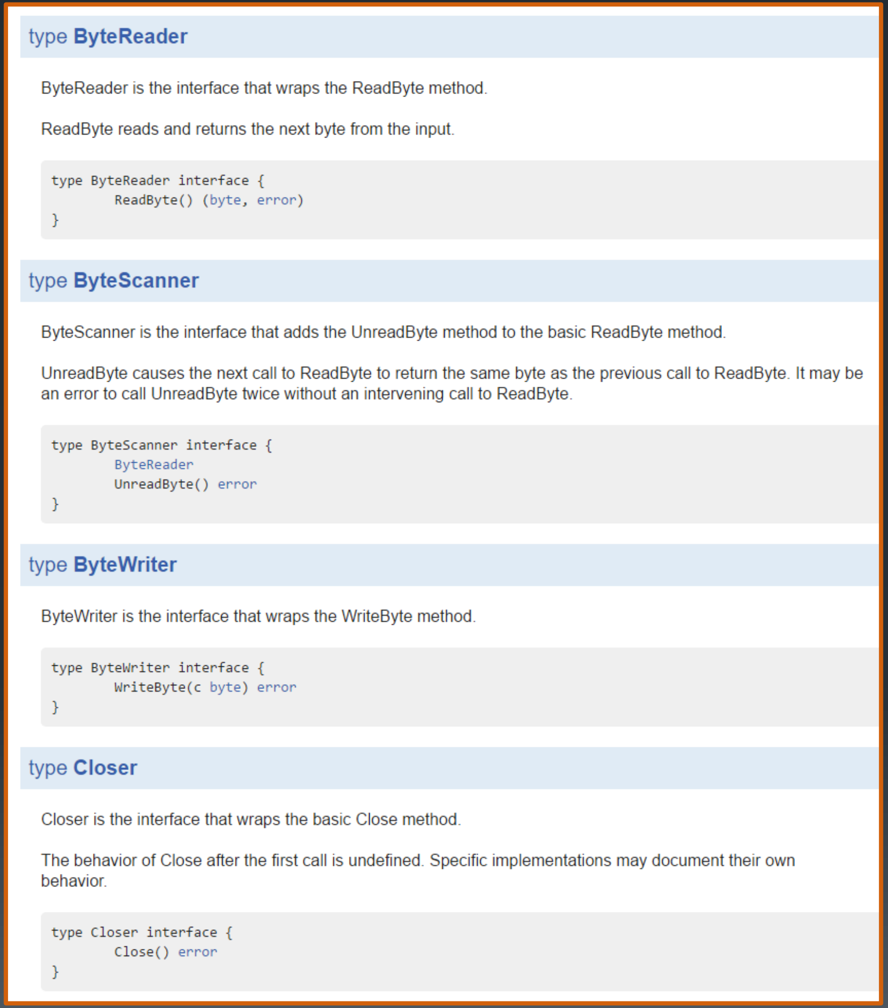
```go
package main

import "fmt"

type Moto struct {
    make  string
    model string
}

func (m Moto) mm() string {
    return m.make + " " + m.model
}

func (m *Moto) ModelSuffix(suf string) {
    m.model += suf
}

func main() {
    bike := Moto{"Honda", "VFR750"}

    bmm := Moto.mm
    fmt.Println(bmm(bike))

    bms := bike.ModelSuffix
    bms("F")
    fmt.Println(bike.mm())
    fmt.Println(bike.make) //we in the same package so this works!
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
2017/04/10 22:50:37 server.go:73: Using API v1
2017/04/10 22:50:37 debugger.go:68: launching process with args: [d:\dev\g
API server listening at: 127.0.0.1:2345
2017/04/10 22:50:37 debugger.go:414: continuing
Honda VFR750
Honda VFR750F
Honda
```

Interfaces

- Interface types express generalizations or abstractions about the behaviors of other types
  - By generalizing, interfaces let us write functions that are more flexible and adaptable because they are not tied to the details of one particular implementation
- Go's interfaces are satisfied implicitly
  - There's no need to declare interfaces that a given type satisfies
  - Simply possessing the necessary methods is enough
  - This is a fairly unique approach to interface/type association
- This design lets you create new interfaces that are satisfied by old types without changing the types
  - particularly useful for types defined in packages that you don't control
- A concrete type specifies the exact representation of its values and exposes the intrinsic operations of that representation
- An Interface type is an abstract type and doesn't expose the representation or internal structure of its values, or the set of basic operations supported
  - It reveals only some of their methods
  - When you have a value of an interface type, you know nothing about what it is; you know only what behaviors are provided by its methods
- The io package offers several key interfaces



type **ByteReader**

ByteReader is the interface that wraps the ReadByte method.

ReadByte reads and returns the next byte from the input.

```
type ByteReader interface {
        ReadByte() (byte, error)
}
```

type **ByteScanner**

ByteScanner is the interface that adds the UnreadByte method to the basic ReadByte method.

UnreadByte causes the next call to ReadByte to return the same byte as the previous call to ReadByte. It may be an error to call UnreadByte twice without an intervening call to ReadByte.

```
type ByteScanner interface {
        ByteReader
        UnreadByte() error
}
```

type **ByteWriter**

ByteWriter is the interface that wraps the WriteByte method.

```
type ByteWriter interface {
        WriteByte(c byte) error
}
```

type **Closer**

Closer is the interface that wraps the basic Close method.

The behavior of Close after the first call is undefined. Specific implementations may document their own behavior.

```
type Closer interface {
        Close() error
}
```

Interface Combinations

- An interface type specifies a set of methods that a concrete type must possess to be considered an instance of that interface
  - A type satisfies an interface if it possesses all the methods the interface requires.
- The io.Writer type is one of the most widely used interfaces because it provides an abstraction of all the types to which bytes can be written
  - Files
  - Memory buffers
  - Network conns
  - HTTP clients
  - Archivers
  - Hashers
- The io package defines interface types in terms of other interfaces as well

type WriteCloser

WriteCloser is the interface that groups the basic Write and Close methods.

```
type WriteCloser interface {
        Writer
        Closer
}
```

type WriteSeeker

WriteSeeker is the interface that groups the basic Write and Seek methods.

```
type WriteSeeker interface {
        Writer
        Seeker
}
```

type Writer

Writer is the interface that wraps the basic Write method.

Write writes len(p) bytes from p to the underlying data stream. It returns the number of bytes written from p (0 <= n <= len(p)) and any error encountered that caused the write to stop early. Write must return a non-nil error if it returns n < len(p). Write must not modify the slice data, even temporarily.

Implementations must not retain p.

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

Implementing an interface

- Imagine you would like to use the fmt.Fprintf method to write formatted text to a buffering type you have
  - The help for Fprintf says it wants an io.Writer which only has one method
  - Implementing that method will satisfy the interface and make it possible for you to use that type with many library functions
- Note that our receiver is a pointer
  - We need to modify the object
- Because of this we must pass the pointer to the object (&b) to functions desiring a Writer interface

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

```
package main

import "fmt"

type StrBuf struct {
    buf string
}

func (sb *StrBuf) Write(p []byte) (n int, err error) {
    sb.buf = string(p)
    return len(sb.buf), nil
}

func main() {
    var b StrBuf
    fmt.Fprintf(&b, "Hi")
    fmt.Println(b.buf)
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
2017/04/10 23:50:05 server.go:73: Using API v1
2017/04/10 23:50:05 debugger.go:68: launching process with args:
API server listening at: 127.0.0.1:2345
2017/04/10 23:50:05 debugger.go:414: continuing
Hi
```

func **Fprintf**

```
func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error)
```

Fprintf formats according to a format specifier and writes to w. It returns the number of bytes written and any write error encountered.

Custom Interfaces

- Interface types are defined with the interface keyword
- Types implement interfaces by offering methods with identical signatures as those in the interface
- Interface variables can point to any object implementing the interface
  - This sets the type and value of the interface variable
    - Known as the dynamic type and value
      - Together these are referred to as the type descriptors
    - The static type of an interface variable is that of the interface
- nil interface calls cause a panic
  - if i == nil //test for unassigned interface variable
- Interface values may be compared using == and !=
  - Two interface values are equal if both are nil, or if their dynamic types are identical and their dynamic values are equal
  - Being comparable, interfaces can be used as map keys

```go
package main

import "fmt"

type Moto struct {
    make, model string
}

func (m Moto) Summary() string {
    return m.make + " " + m.model
}

type Veh interface {
    Summary() string
}

func main() {
    bike1 := Moto{"Honda", "VFR750"}
    bike2 := Moto{"Ducati", "Paso750"}
    var i Veh
    i = bike1
    fmt.Println(i.Summary())
    i = bike2
    fmt.Println(i.Summary())
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
2017/04/11 00:14:54 server.go:73: Using API v1
2017/04/11 00:14:54 debugger.go:68: launching pr
API server listening at: 127.0.0.1:2345
2017/04/11 00:14:54 debugger.go:414: continuing
Honda VFR750
Ducati Paso750
```

Type Assertion

- Type assertions in Go allow you to test the types supported by the dynamic type associated with an interface
  - Form: i.(T)
    - Where i is an interface and T is the desired type
  - Type assertions return a tuple with the desired interface type and an ok status
    - i2, ok := i.(T)
    - The type tested need NOT be the current dynamic type of i or a component thereof
- Much like dynamic casting in other languages
  - A key difference is that the interface supplied and the interface derived need not be related in any way

```go
package main

import "fmt"

type Moto struct {
    make, model string
}

func (m Moto) Summary() string {
    return m.make + " " + m.model
}

type Veh interface {
    Summary() string
}

type Car interface {
    Drive() string
}

func main() {
    bike := Moto{"Honda", "VFR750"}
    var i Veh
    i = bike
    if v, ok := i.(Veh); ok {
        fmt.Println(v.Summary())
    }
    if c, ok := i.(Car); ok {
        fmt.Println(c.Drive())
    }
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
2017/04/11 00:32:57 server.go:73: Using API v1
2017/04/11 00:32:57 debugger.go:68: launching proc
API server listening at: 127.0.0.1:2345
2017/04/11 00:32:57 debugger.go:414: continuing
Honda VFR750
```

Go Tips

- When designing a new package:
  - Use Interfaces when two or more concrete types must be dealt with in a uniform way
  - Avoid single implementation interfaces
    - These are unnecessary abstractions and have a run-time cost
    - The exception is a single type interface that cannot live in the same package as the type because of dependencies (an interface can decouple the two packages)
  - Keep interfaces minimal, crisp and generic
    - Because interfaces are used when implemented by multiple types they tend to have fewer, simpler methods (often just one)
    - Small interfaces are easier to satisfy when new types come along
    - A good rule of thumb for interface design is ask only for what you need
  - Control method visibility outside a package through exporting

Source: Donovan/Kernighan

Summary

- Go support several key OO features:
  - Methods
  - Receivers
  - Method binding
  - Encapsulation
  - Interfaces
  - Type Assertions

Lab: Methods and Interfaces

- Working with methods and interfaces