Module 4: Interfaces for Abstraction
Topic 1.1: Polymorphism

Polymorphism

- Ability for an object to have different “forms” depending on the context
- Example: `Area()` function
  - Rectangle, **area = base * height**
  - Triangle, **area = 0.5 * base * height**
- **Identical** at a high level of abstraction
- **Different** at a low level of abstraction

Inheritance

- Subclass inherits the methods/data of the superclass
- Example: **Speaker** superclass
  - `Speak()` method, prints “<noise>”
- Subclasses **Cat** and **Dog**
  - Also have the `Speak()` method
- Cat and Dog are different forms of Speaker
- Remember: Go does not have inheritance

Overriding

- Subclass **redefines a method** inherited from the superclass
- Example: Speaker, Cat, Dog
  - Speaker `Speak()` prints “<noise>”
  - Cat `Speak()` prints “meow”
  - Dog `Speak()` prints “woof”
- `Speak()` is polymorphic
  - Different implementations for each class
  - Same **signature** (name, params, return)

Module 4: Interfaces for Abstraction
Topic 1.2: Interfaces

Interfaces

- Set of **method signatures**
  - Name, parameters, return values
  - Implementation is NOT defined
- Used to express conceptual similarity between types
- Example: **Shape2D interface**
- All 2D shapes must have `Area()` and `Perimeter()`

Satisfying an Interface

- Type **satisfies an interface** if type defines all methods specified in the interface
  - Same method signatures
- **Rectangle** and **Triangle** types satisfy the **Shape2D** interface
  - Must have `Area()` and `Perimeter()` methods
  - Additional methods are OK
- Similar to inheritance with overriding

Defining an Interface Type

```
type Shape2D interface {
   Area() float64
   Perimeter() float64
}
type Triangle {…}
func (t Triangle) Area() float64 {…}
func (t Triangle) Perimeter() float64 {…}
```

- Triangle type satisfies the Shape2D interface
- No need to state it explicitly

Module 4: Interfaces for Abstraction
Topic 1.3: Interface vs. Concrete Types

Concrete vs Interface Types

Concrete Types

- Specify the exact representation of the data and methods
- Complete method implementation is included

Interface Types

- Specifies some method signatures
- Implementations are abstracted

Interface Values

- Can be treated like other values
  - Assigned to variables
  - Passed, returned
- Interface values have two components

1. **Dynamic Type**: Concrete type which it is assigned to
2. **Dynamic Value**: Value of the dynamic type

- Interface value is actually a pair
  - **(dynamic type, dynamic value)**

Defining an Interface Type

```
type Speaker interface {Speak ()}

type Dog struct {name string}
func (d Dog) Speak() {
   fmt.Println(d.name)
}
func main() {
   var s1 Speaker
   var d1 Dog{“Brian”}
   s1 = d1
   s1.Speak()
}
```

- Dynamic type is Dog, Dynamic value is d1

Interface with Nil Dynamic Value

- An interface can have a nil dynamic value

```
var s1 Speaker
var d1 *Dog
s1 = d1
```

- d1 has no concrete value yet
- s1 has a dynamic type but no dynamic value

Nil Dynamic Value

- Can still call the `Speak()` method of s1
- Doesn’t need a dynamic value to call
- Need to check inside the method

```
func (d *Dog) Speak() {
   if d == nil {
      fmt.Println(“<noise>”)
   } else {
      fmt.Println(d.name)
   }
}
var s1 Speaker
var d1 *Dog
s1 = d1
s1.Speak()
```

Nil Interface Value

- Interface with **nil dynamic type**
- Very different from an interface with a **nil dynamic value**

Nil dynamic value and **valid dynamic type**

- Can call a method since type is known

```
var s1 Speaker
var d1 *Dog
s1 = d1
```

Nil dynamic type

- Cannot call a method, runtime error

```
var s1 Speaker
```

Module 4: Interfaces for Abstraction
Topic 2.1: Using Interfaces

Ways to Use an Interface

- Need a function which takes multiple types as a parameter
- Function `foo()` parameter
  - Type X or type Y
- Define interface Z
- `foo()` parameter is interface Z
- Types X and Y satisfy Z
- Interface methods must be those needed by `foo()`

Pool in a Yard

- I need to put a pool in my yard
- Pool needs to fit in my yard
  - Total area must be limited
- Pool needs to be fenced
  - Total perimeters must be limited
- Need to determine if a pool shape satisfies criteria
- **`FitInYard()`**
  - Takes a shape as a argument
  - Returns true if the shape satisfies criteria

FitInYard()

- Many possible shape types
  - Rectangle, triangle, circle, etc.
- `FitInYard()` should take many shape types
- Valid shape types must have:
  - `Area()`
  - `Perimeter()`
- Any shape with these methods is OK

Interface for Shapes

```
type Shape2D interface {
   Area() float64
   Perimeter() float64
}
type Triangle {…}
func (t Triangle) Area() float64 {…}
func (t Triangle) Perimeter() float64 {…}
type Rectangle {…}
func (t Rectangle) Area() float64 {…}
func (t Rectangle) Perimeter() float64 {…}
```

- Rectangle and Triangle satisfy Shape2D interface

FitInYard() Implementation

```
func FitInYard(s Shape2D) bool {
   if (s.Area() < 100 &&
       s.Perimeter() < 100) {
     return true
   }
   return false
}
```

- Parameter is any type that satisfies the interface

Empty Interface

- Empty interface specifies no methods
- All types satisfy the empty interface
- Use it to have a function accept any type as a parameter

```
func PrintMe(val interface{}) {
   fmt.Println(val)
}
```

Module 4: Interfaces for Abstraction
Topic 2.2: Type Assertions

Concealing Type Differences

- Interfaces hide the differences between types

```
func FitInYard(s Shape2D) bool {
  if (s.Area() < 100 &&
     s.Perimeter() < 100) {
    return true
  }
  return false
}
```

- Sometimes you need to treat different types in different ways

Exposing Type Differences

- Example: Graphics program
- **`DrawShape()`** will draw any shape
  - `func DrawShape(s Shape2D) { …`
- Underlying API has different drawing functions for each shape
  - `func DrawRect(r Rectangle) { …`
  - `func DrawTriangle(t Triangle) { …`
- Concrete type of shape s must be determined

Type Assertions

- Type assertions can be used to determine and extract the underlying concrete type

```
func DrawShape(s Shape2D) bool {
   rect, ok := s.(Rectangle)
```

- Type assertion extracts Rectangle from Shape2D
  - Concrete type in parentheses
- If interface contains concrete type
  - rect == concrete type, ok == true
- If interface does not contain concrete type
  - rect == zero, ok == false

Type Assertions for Disambiguation

```
func DrawShape(s Shape2D) bool {
   rect, ok := s.(Rectangle)
   if ok {
      DrawRect(rect)
   }
   tri, ok := s.(Triangle)
   if ok {
      DrawTriangle(tri)
   }
}
```

Type Switch

- Switch statement used with a type assertion

```
func DrawShape(s Shape2D) bool {
   switch sh := s.(type) {
   case Rectangle:
      DrawRect(sh)
   case Triangle:
      DrawTriangle(sh)
   }
}
```

Module 4: Interfaces for Abstraction
Topic 2.3: Error Handling

Error Interface

- Many Go programs return error interface objects to indicate errors

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

- Correct operation: error == nil
- Incorrect operation: `Error()` prints error message

Handling Errors

- Check whether the error is nil
- If it is not nil, handle it

```
f, err := os.Open(“/harris/test.txt”)
if err != nil {
   fmt.Println(err)
   return
}
```

- `fmt` package calls the `Error()` method to generate string to print