Lecture 17: Interfaces

The principle of encapsulation

A fundamental design principle in programming is encapsulation:

- group together related things, and hide as many details as possible from the rest of the world, exposing only a small "interface" to the rest of the program.

Examples we have seen so far:

- **Functions** — to use "fmt.Printf" I only need to know the rules about what parameters it takes and what it returns; how it is implemented is totally hidden from me.

- **Packages** — inside the "fmt" package is a huge amount of code, but we only need to know about the functions.

- **Objects & methods** — I access a stack using only the methods on the Stack type --- it doesn't matter if Stack is implemented as a list or some other technique. \( \text{S.Pop()} \) and \( \text{S.Push(x)} \) hide those details, but still let you share data between invocations of functions.

Interfaces

Interfaces let you formally define a set of operations that a type supports. They let you hide completely how the operations are carried out. They let you specify a type that will support a given set of operations without even specifying the details about the type.

This will be made clearer with an example.

**Example: Design for a drawing program**

**Drawing program objects**

A typical drawing program manipulates: shapes, text, lines. It also displays and allows users to manipulate handles on the shapes, colors, shadows, layers, canvases, etc.
It would be natural to create an object for each type of shape: Circle, Oval, Triangle, Star, Square, ....

For example, here’s an object for Square:

```
type Square struct {
    x0, y0 int
    x1, y1 int
    fillColor color.Color
    strokeColor color.Color
    lineWidth int
}

func (s *Square) MoveTo(x, y int)
func (s *Square) Resize(w, h int)
func (s *Square) Handles() []Handles
func (s *Square) Draw(c *DrawingCanvas)
func (s *Square) SetLineWidth(w int)
func (s *Square) ContainsPoint(x, y int)
```
And here’s an object for Oval:

```go
type Oval struct {
    x0, y0 int
    radius int
    fillColor color.Color
    strokeColor color.Color
    lineWidth int
}
```

The methods for Square and Oval are needed for every shape.

**Challenge: a DrawingCanvas type**

We want to write a type that corresponds to a canvas. A canvas can have lots of different shapes on it:

```go
type DrawingCanvas struct {
    width, height int
    backgroundColor color.Color
    shapes [][]???    // <- what type should go here!!!?
}
```

Question 1: What type should the shape field have?

We assume that DrawingCanvas should have a method to draw all the shapes:

```go
func (c *DrawingCanvas) DrawAllShapes()
```

This function should should call the Draw() function on each of the shapes that the canvas contains. It should do something like:
Question 2: How can the above code know to call 
(*Oval) Draw for ovals and (*Square) Draw for squares?

The benefits of the above design are that:

- **DrawAllShapes** is conceptually very simple: it just loops through the shapes and asks each of them to draw themselves.

- All the shape-specific knowledge is encapsulated inside each shape type: an Oval knows how to draw itself; a Square knows how to draw itself, etc.

- Adding a new shape is easy: just create a new shape type. You don't need to modify any existing shape types (each shape can store the data it needs, i.e. radius vs. width/length). You don't even need to modify **DrawAllShapes** when you add a shape!

So how do we answer the above 2 questions so we can use this design? The answer to both of these questions is the use of **interface** types.

```go
func (c *DrawingCanvas) DrawAllShapes() {
    for shape := range shapes {
        shape.Draw(c)
    }
}
```

The main problem above is that the shapes all have different types but we want to put them into a single list.

The thing that is common to "shapes" is what you can do with them: **Draw**, **MoveTo**, **Resize**, etc.

Go lets you define a type that specifies only the operations that can be performed on the type:

```go
type Shape interface {
    MoveTo(x, y int)
    Resize(w, h int)
    Handles() []Handles
    Draw(c *DrawingCanvas)
    SetLineWidth(w int)
}
```

This looks nearly the same as a **struct** but (a) using the word **interface** and (b) listing **functions**
instead of data. The way to read this is that a Shape is a thing that has these methods.

If I have a Shape, I don’t need to know what kind of shape, or how its shape functions are implemented.

**Modifying the DrawingCanvas struct:**

Now we can write:

```go
type DrawingCanvas struct {
    width, height int
    backgroundColor color.Color
    shapes []Shape
}
```

This makes the shape field be a list of Shapes. This list can now contain anything that supports all the methods of the Shape interface.

```go
func (c *DrawingCanvas) DrawAllShapes() {
    for shape := range shapes {
        // we know shape as a Draw() method b/c it is a Shape
        shape.Draw(c)
    }
}
```

**Putting it all together**

Here are some Shape objects:
We can then write the functions:
Duck typing

Notice that we never said that Oval was a Shape or that Square was a Shape. Go figured that out on its own.

This is called "duck typing" because "If it walks like a duck, swims like a duck, and quacks like a duck, it's a duck." So "If it Draw()s like a Shape, MoveTo()s like a Shape, and Resize()s like a Shape, it's a Shape."

Summary
- Interfaces let you create a type that depends only on the operations you can perform on it.

- Let's you write general code that words for any type that supports that interface.

Glossary

- **encapsulation**: the principle that programs should be "local": details should be hidden and any dependencies should be confined to small parts of a program.