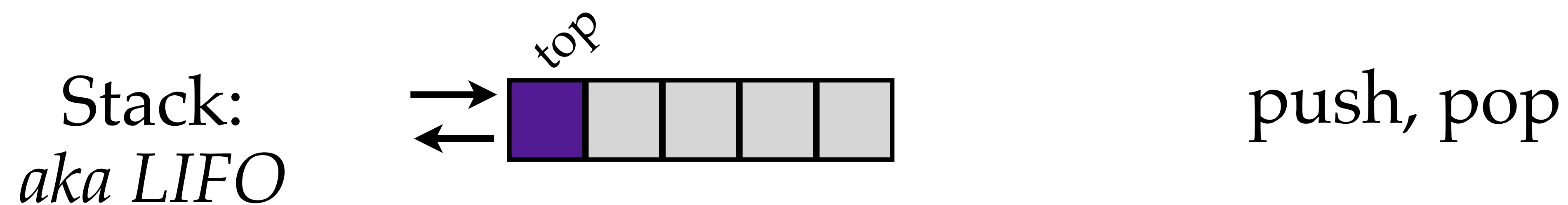
More Object-Oriented Programming: Encapsulation, Interfaces

02-201 / 02-601

Example 1: A Stack

Designing a Stack the Object-Oriented Way



LIFO = last-in, first-out

- Our "noun" is the stack
- Our "verbs" are push, pop, create

Recall: Non-OO implementation

```
func createStack() []int {
   return make([]int, 0)
}
```

```
func push(S []int, item int) []int {
   return append(S, item)
}
```

```
func pop(S []int) ([]int, int) {
   if len(S) == 0 {
       panic("Can't pop empty stack!")
   }
   item := S[len(S)-1]
   S = S[0:len(S)-1]
   return S, item
}
```

```
func main() {
     S := createStack()

     S = push(S, 1)
     S = push(S, 10)
     S = push(S, 13)
     fmt.Println(S)

     S, item := pop(S)
     fmt.Println(item)

     S, item = pop(S)
     fmt.Println(item)

     S, item = pop(S)
     fmt.Println(item)
}
```

Object Oriented Implementation:

```
type Stack struct {
    items []int
}
```

Step 1: Define a type that corresponds to our noun that can hold the data we need for a stack

Step 2: Define *methods* for the verbs: Push, Pop:

```
func (S *Stack) Push(a int) {
    S.items = append(S.items, a)
}

func (S *Stack) Pop() int {
    a := S.items[len(S.items)-1]
    S.items = S.items[:len(S.items)-1]
    return a
}
```

Define a regular function for Create:

- In order to call a method, need a variable of the appropriate type.
- So: “Create” can’t be a method since that is how we create a variable of this type.

```
func CreateStack() Stack {
    return Stack{items: make([]int, 0)}
}
```

- Sometimes called a “factory function” since it creates variables of a given type.

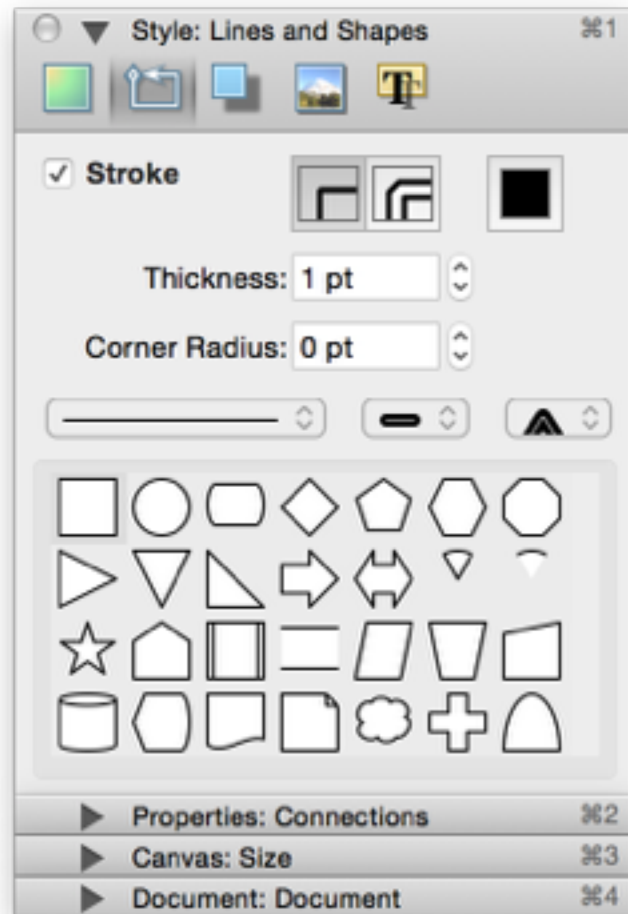
Using the Stack

- Now much nicer because we don’t have to also return the new stack:

```
S := CreateStack()
S.Push(10)
S.Push(20)
fmt.Println(S.Pop())
```

Example 2: A Drawing Program

Design for A Drawing Program

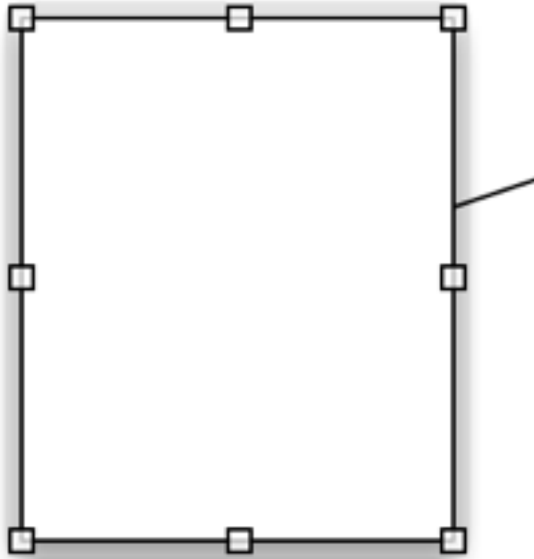


- A typical drawing program (this one is OmniGraffle)
- Manipulates: shapes, text, lines
- Also: handles on the shapes, colors, shadows, layers, canvases, etc.

Canvas

Shapes

Shapes

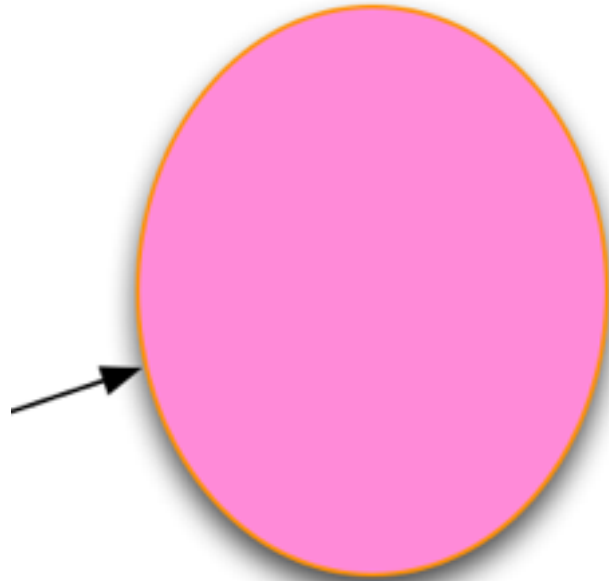


```
type Square struct {
    x0,y0 int
    x1,y1 int
    fillColor color.Color
    strokeColor color.Color
    lineWidth int
}
```

- Natural to create an object type for each shape:
  - Circle
  - Oval
  - Triangle
  - Star
  - Square
  - ….

```
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)
```

Shapes



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

- Natural to create an object type for each shape:
  - Circle
  - Oval
  - Triangle
  - Star
  - Square
  - ….

```
func (s *Oval) MoveTo(x,y int)
func (s *Oval) Resize(w,h int)
func (s *Oval) Handles() []Handles
func (s *Oval) Draw(c *DrawingCanvas)
func (s *Oval) SetLineWidth(w int)
func (s *Oval) ContainsPoint(x,y int)
```

These functions are needed for ***all*** shapes.

DrawingCanvas

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

What type can go here ???? if our canvas may contain Squares, Circles, Triangles?

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

Should call the “Draw()” function on each of the shapes the canvas contains

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

Before Solving the Problem: The benefits of this design

- DrawAllShapes is conceptually very simple:
  - just loop through the shapes and ask each of them to draw themselves
- All the shape-specific knowledge is embedded 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
  - Don’t need to modify any existing shape types (each shape can store the data it needs, i.e. radius vs. width/length)
  - Don’t need to modify DrawAllShapes!

interface{}

- The problem above is that the shapes all have different types but we want to put them into a single slice.
- 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 possible methods:

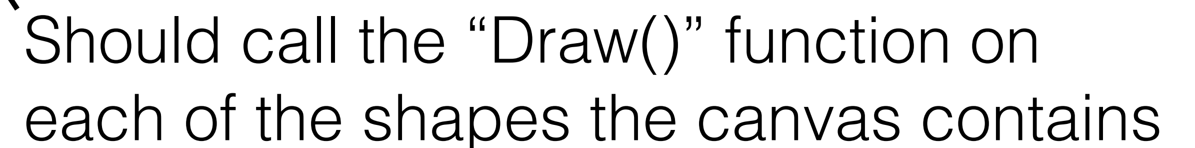
```
type Shape interface {
   MoveTo(x,y int)
   Resize(w,h int)
   Handles() []Handles
   Draw(c *DrawingCanvas)
   SetLineWidth(w int)
}
```

Means: a Shape is a thing that has these methods

DrawingCanvas — with Interface

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

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




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

Simplified Drawing Example

```go
//=================================
// What all shapes must do
//=================================

type Shape interface {
   MoveTo(x,y int)
   Draw()
}

//=================================
// An Oval Shape
//=================================

type Oval struct {
   x0,y0 int
}

func (s *Oval) MoveTo(x,y int) {
    s.x0, s.y0 = x,y
}

func (s *Oval) Draw() {
    fmt.Println("I'm an OVAL!!!! at", s.x0, s.y0)
}

//=================================
// A Square Shape
//=================================

type Square struct {
   x0,y0 int
}

func (s *Square) MoveTo(x,y int) {
    s.x0, s.y0 = x,y
}

func (s *Square) Draw() {
    fmt.Println("I'm a SQUARE!!!! at ", s.x0, s.y0)
}
```

```go
//=================================
// A function to draw all the shapes
//=================================

func DrawAllShapes(shapes []Shape) {
    fmt.Println("===================================")
    for _, shape := range shapes {
        shape.Draw()
    }
    fmt.Println("===================================")
}

//=================================
// Create some shapes and add them to the list
//=================================

func main() {
    shapes := make([]Shape, 0)
    var s1 Shape = &Square{10,10}
    var s2 Shape = &Square{100,100}
    var s3 Shape = &Oval{60,75}
    shapes = append(shapes, s1)
    shapes = append(shapes, s2)
    shapes = append(shapes, s3)

    DrawAllShapes(shapes)

    shapes[1].MoveTo(3333,3333)
    DrawAllShapes(shapes)

}
```

Duck typing

Note: we never explicitly said that Square or Oval were Shapes!

Luis Miguel Bugallo Sánchez (Lmbuga Commons)(Lmbuga Galipedia)

"If it walks like a duck, swims like a duck, and quacks like a duck, it's a duck."

If it Draw()s like a Shape, MoveTo()s like a Shape, and Resize()s like a Shape, it's a Shape.

Interfaces & Pointers

- An interface is a set of methods that can be called on the type.
- Our methods are expecting a * type:

```
func (s *Oval) Draw() {
    fmt.Println("I'm an OVAL!!!! at", s.x0, s.y0)
}
```

- So we store a pointer to the shape inside our Shape variable:

```
var s1 Shape = &Square{10,10}
```

- Note though: s1 is *not a pointer:* It’s a variable of an interface type that holds a pointer to the thing that satisfies the interface.

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
  - expose only a small “interface” to the rest of the program.
- Examples:
  - **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.
  - **Interfaces** — if I have a Shape, I don’t need to know what kind of shape, or how its shape functions are implemented.

Summary

- Create interfaces if you have a number of related “nouns” that will all do the same thing
- You can declare variables of the type of the interface that can hold any variable that supports that interface’s methods.
- Let’s you write general code that depends only on the methods that you expect to exist.