UCI Division of Continuing Education

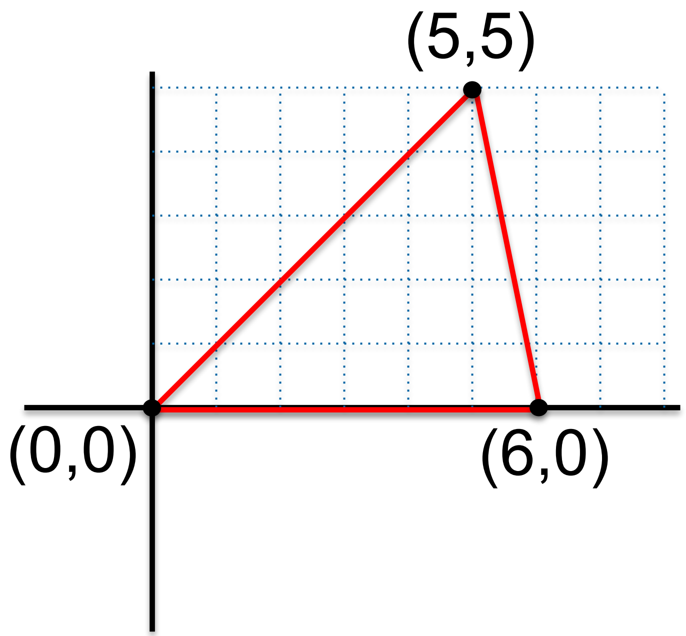
Module 3: Object-Orientation in Go
Topic 1.1: Classes and Encapsulation

Classes

- Collection of data fields and functions that share a well-defined responsibility
- Example: **Point** class
  - Used in a geometry program
  - Data: x coordinate, y coordinate
  - Functions:
    - `DistToOrigin(), Quadrant()`
    - `AddXOffset(), AddYOffset()`
    - `SetX(), SetY()`
- Classes are a **template**
- Contain **data fields**, not data

Object

- Instance of a class
- Contains real data
- Example: Point class



Encapsulation

- Data can be protected from the programmer
- Data can be accessed only using methods
- Maybe we **don't trust the programmer** to keep data consistent
- Example: Double distance to origin
  - Option 1: Make method `DoubleDist()`
  - Option 2: Trust programmer to double X and Y directly

Module 3: Object-Orientation in Go
Topic 1.2: Support for Classes

No “Class” Keyword

- Most OO languages have a class keyword
- Data fields and methods are defined inside a class block

```
class Point:
    def __init__(self, xval, yval):
        self.x = xval
        self.y = yval
```

Associating Methods with Data

- Method has a **receiver type** that it is associated with
- Use dot notation to call the method

```
type MyInt int

func (mi MyInt) Double () int {
   return int(mi*2)
}
func main() {
   v := MyInt(3)
   fmt.Println(v.Double())
}
```

Implicit Method Argument

```
func (mi MyInt) Double () int {
   return int(mi*2)
}
func main() {
   v := MyInt(3)
   fmt.Println(v.Double())
}
```

- Object v is an implicit argument to the method
- Call by value

Module 3: Object-Orientation in Go

Topic 1.3: Support for Classes

Structs, again

- Struct types compose data fields

```
type Point struct {
   x float64
   y float64
}
```

- Traditional feature of classes

Structs with Methods

- **Structs and methods** together allow arbitrary data and functions to be composed

```
func (p Point) DistToOrig() {
   t := math.Pow(p.x, 2) +
math.Pow(p.y, 2)
   return math.Sqrt(t)
}
func main() {
   p1 := Point(3, 4)
   fmt.Println(p1.DistToOrig())
}
```

Encapsulation in Go

- Making data fields or methods hidden from the programmer
- Might use a `private` keyword in another language
- Example: Point struct, `Scale()` method
- `Scale()` should multiply x and y coordinates by a constant
- Don’t trust this to the programmer
  - Might scale one coordinate but not the other
  - Coordinates could become inconsistent
  - Need to **hide x and y coordinates**

Hiding in a Package

- Go can only hide data/methods in a package
- Variables/functions are only exported if their names start with a **capital letter**

```
package data
var x int = 1
var Y int = 2
```

```
package main
import “data"
func main() {
   fmt.Println(Y)
   fmt.Println(x)
}
```

Module 3: Object-Orientation in Go
Topic 2.1: Encapsulation

Controlling Access

- Can define **public functions** to allow access to hidden data

```
package data
var x int = 1
func PrintX() {fmt.Println(x)}
```

```
package main
import “data"
func main() {
   data.PrintX()
}
```

Controlling Access to Structs

- Hide fields of structs by starting field name with a lower-case letter

```
package data
type Point struct {
   x float64
   y float64
}
func (p *Point) InitMe(xn, yn float64) {
   p.x = xn
   p.y = yn
}
```

- Need `InitMe()` to assign hidden data fields

Controlling Access to Structs

- Define public methods which access hidden data

```
func (p *Point) Scale(v float64) {
  p.x = p.x * v
  p.y = p.y * v
}
func (p *Point) PrintMe(){
  fmt.Println(p.x, p.y)
}
```

Controlling Access to Structs

- Access to hidden fields only through public methods

```
package main
func main() {
   var p data.Point
   p.InitMe(3, 4)
   p.Scale(2)
   p.PrintMe()
}
```

Module 3: Object-Orientation in Go
Topic 2.2: Pointer Receivers

Limitations of Methods

- Receiver is passed implicitly as an argument to the method
- Method cannot modify the data inside the receiver
- Example: `OffsetX()` should increase x coordinate

```
func main() {
   p1 := Point(3, 4)
   p1.OffsetX(5)
}
```

Large Receivers

- If receiver is large, lots of copying is required

```
type Image [100][100]int
func main() {
    i1 := GrabImage()
    i1.BlurImage()
}
```

- 10,000 ints copied to `BlurImage()`

Pointer Receivers

- Receiver can be a pointer to a type
- Call by reference, pointer is passed to the method

```
func (p *Point) OffsetX(v float64)
{
  p.x = p.x + v
}
```

Module 3: Object-Orientation in Go
Topic 2.3: Pointer Receivers, Referencing, Dereferencing

No Need to Dereference

- Point is referenced as p, not *p
- Dereferencing is automatic with . operator

```
func (p *Point) OffsetX(v int) {
   p.x = p.x + v
}
```

No Need to Reference

- Do not need to reference when calling the method

```
func main() {
   p := Point{3, 4}
   p.OffsetX(5)
   fmt.Println(p.x)
}
```

Using Pointer Receivers

- Good programming practice:

1. All methods for a type have **pointer receivers**, or
2. All methods for a type have **non-pointer receivers**

- Mixing pointer/non-pointer receivers for a type will get confusing
  - Pointer receiver allows modification