Go for Game Programmers

Go Headline features

- Go (Sometimes calls Golang)
  - Headline features:
    - Compiled
    - Statically typed
      - Variable will always refer to same type of value
    - Structurally Typed
      - Type equivalence by definition not name
    - Memory safe
      - No buffer overflows, unsafe pointer operations
    - Garbage collected
    - Focus on concurrency
      - One of Go’s claims to fame – but less important to early game work

First Imressions of Go

- When I first looked at Go
  - It looked like python and C++ had a baby
  - Of course I don’t know algol
- Python philosophy :
  - There is one ‘right’ way to do things
  - (harder to see recently)
  - This is pythonic, but not enforced by compiler/interpreter.
- With go – often is enforced by compiler

Good Style

- Go ‘Good Style’ is often compiler enforced
  - Unused local variables are a compiler error
  - Unused imports are a compiler error
- Online flame wars are often about what “good style” is
- Go often settles these by making the compiler only work for the approved style
- Go helps you be compile ready with gofmt
  - Can run on command line
    - Or let goland do it for you.
  - Gofmt pronunciation
    - Do you go with the majority?
    - Or with the crusading minority?
  - Gofmt sort of like python-black
    - Simply rewrites your code to be ‘proper’ (idiomatic) go

Go Programming

- Let’s look at a few basic programming concepts in go

Comments and Types

- Comments are really useful when learning a language
- Comments in Go same as C++
  - Go took them just like java did
  - // line comments
  - /*
  - Multiline comments
  - */
  -

- Go, like java, has distinction between basic types and all other types
- Basic types:
  - boolean
  - string
  - and number (several number types)
    - uint8, uint16, uint32, uint64, int8, int16, int32 and int64, etc

Variables and

- Variables are statically typed, but type can be inferred
  - var name string //creates a new variable called name of type string with an empty string
  - var name2 = “Imelda” //creates a new variable called name2 of type string with initial value “Imelda”
  - var num1, num2 int = 100, 300
  - var3 :=3.14159

- In the other column, var3 is clearly what (not the type but the value)?

Variables and Constants

- Variables are statically typed, but type can be inferred
  - var name string //creates a new variable called name of type string with an empty string
  - var name2 = “Imelda” //creates a new variable called name2 of type string with initial value “Imelda”
  - var num1, num2 int = 100, 300
  - var3 :=3.14159

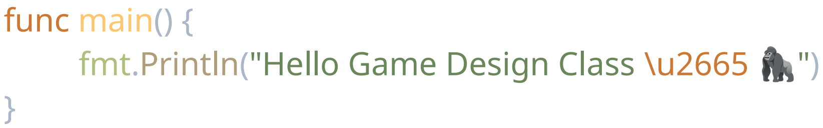
- In the other column, var3 is clearly what?
- Pi right? So it really shouldn’t be a variable

- Constants in go, more like C++ than python
  - const pi = 3.14159 //math.Pi is better
  - can’t be changed
  - Notice constant is mixed case
    - Most languages have upper case.
    - Why? (class discussion)

Functions

- Create a function in go using keyword func,
  - func <function name>(<param list>) <ret type>{
    - <function body>
    - }
  - A few things to look at here:
    - Return type is after param list (unlike java/c/C++, but like python/swift)
    - Param list can be empty, when not, param name first then type
    - And that opening brace? It must be there. Compile error for being on next line.
      - Avoids one of the favorite java flame wars

- Example



```
func main() {
    fmt.Println("Hello Game Design Class \u2665 🦍")
}
```

  - Main function in main package is entry point of go program
  - fmt package has lots of functions to print and build strings.
  - Strings can contain emoji ‘runes’ (characters) natively
- Function returning value
  - `func add(x int, y int) int {`
    - `return x + y`
  - `}`

Import

- Like most languages, if you want code beyond the always available basics, must import
- In previous slide imports to needed use code from other packages (like fmt)
  - And their exported symbols
- Import a single package
  - import “fmt”
- More commonly, import multiple packages

```
import (
   "fmt"
   "log"
   "net/http"
)
```

- Important: an unused imported library is a compile error
  - gofmt to the rescue – run automatically by goland

Structs

- Go doesn’t have classes, it has structs
  - If you squint hard enough they look like c-structs
  - A collection of named typed fields
  - Eg:

  -
    ```
    type Player struct {
       name string
       health int
       jumpDistance int
    }
    ```
  - Creates a struct type with 3 fields,
    - `var player1 Player //creates a variable player1 of type Player with zero value for the fields`
  - Access fields in a C-like manner
    - `player1.jumpDistance = 3`

- Another example struct

```
type firstGame struct {
        player *ebiten.Image
        xloc int
        yloc int
        score int
}
```

Methods

- What is the difference between methods and function in a language like python or java?

Methods

- What is the difference between methods and function?
  - You call methods on an object
  - You just call functions with parameters.
- Go has methods, you call them on struct objects
- Unlike these other languages, you don't need to write all of the methods for a type together
  - But maybe you should.

- Method syntax
  - func (object) <function name> (<parameter list) (return list){
    - Function body
  - }
- For example

```
// Layout will return the screen dimensions.
func (g *Game) Layout(width, height int) (int, int) {

   return width, height }
```

Interfaces

- Interfaces in go work *kinda* like those in java
  - In that they specify a set of methods that must be implemented to implement the interface
  - But can be written anywhere in the package.

- This is a copy of the ebiten.Game interface from the library
  - (with all of the original comments removed so it fits on the slide)

```
type Game interface {

Update() error

Draw(screen *ebiten.Image)

Layout(outsideWidth, outsideHeight int) (screenWidth, screenHeight int)
}
```

  - Any struct with these three methods can be used as a game object

The beginning of our first Game

```go
package main

import (
	"fmt"
	"github.com/hajimehoshi/ebiten/v2"
	"golang.org/x/image/colornames"
)
type firstGame struct {
	player *ebiten.Image
	xloc   int
	yloc   int
	score  int
}
func main() {
	ebiten.SetWindowSize(1000, 1000)
	ebiten.SetWindowTitle("First Class Example")
	ourGame := firstGame{} //we will use the zero value for now
	err := ebiten.RunGame(&ourGame)
	if err != nil {
		fmt.Println("Failed to run game", err)
	}
}
```

Let's walk through the code here
And on the next slide so we all
understand it.

So let’s try to make that minimal Game

- Lets put together the simplest of these three
- Let’s go over here – and run it.

```
func (game *firstGame) Update() error {
	return nil
}

func (game *firstGame) Draw(screen *ebiten.Image) {
	screen.Fill(colornames.Blanchedalmond)
}

func (game firstGame) Layout(outsideWidth, outsideHeight int) (screenWidth, screenHeight int) {
	return outsideWidth, outsideHeight //by default, just return the current dimensions
}
```

Ok – now what

- Ok, that was our ebitengine
'Hello world" now what
- 2D games are what at their root?

Ok – now what

- Ok, that was our ebitengine 'Hello world" now what
- 2D games are what at their root?
  - Draw some images
  - Move some of the images around on the screen (according to the game design)
  - See if any of them overlap
  - So first lets get an image we can use

- The easiest way to get an image
- ebitenutil.NewImageFromFile
  - We'll use more robust approaches later
  - Get the png image we'll use for this demo
  - We need to load the png image processing library (even though we don't use it NewImage from file does)
  - Add to your imports

```
_ "image/png"
```

that underscore is important!!!!

Main with an image loaded

```
func main() {
	ebiten.SetWindowSize(1000, 1000)
	ebiten.SetWindowTitle("First Class Example")
	//New image from file returns image as image.Image (_) and ebiten.Image
	playerPict, _, err := ebitenutil.NewImageFromFile("ship.png")
	if err != nil {
		fmt.Println("Unable to load image:", err)
	}
	ourGame := firstGame{player: playerPict,
		xloc: 500, yloc: 500}
	err = ebiten.RunGame(&ourGame)
	if err != nil {
		fmt.Println("Failed to run game", err)
	}
}
```

- Lets look at it and ask questions
- Then run it (and ask more questions)

Main with an image loaded

```go
func main() {
        ebiten.SetWindowSize(1000, 1000)
        ebiten.SetWindowTitle("First Class Example")
        //New image from file returns image as image.Image (_) and ebiten.Image
        playerPict, _, err := ebitenutil.NewImageFromFile("ship.png")
        if err != nil {
                fmt.Println("Unable to load image:", err)
        }
        ourGame := firstGame{player: playerPict,
                xloc: 500, yloc: 500}
        err = ebiten.RunGame(&ourGame)
        if err != nil {
                fmt.Println("Failed to run game", err)
        }
}
```

- Lets look at it and ask questions
- Then run it
- Aaand we see nothing has changed

Draw

- So now let's draw the image
- First we create the draw options struct
  - We can use one to draw multiple images in the future, so traditionally call reset before each one
  - Then translate the options to the image location
  - Finally draw the image on the screen
- Let's see it work

```
func (game *firstGame) Draw(screen *ebiten.Image) {
	screen.Fill(colornames.Blanchedalmond)
	drawOps := ebiten.DrawImageOptions{}
	drawOps.GeoM.Reset()
	drawOps.GeoM.Translate(float64(game.xloc), float64(game.yloc))
	screen.DrawImage(game.player, &drawOps)
}
```

GameLoop

- In computer games there is always a 'game loop'
  - Sometimes the library/framework provides it
    - unity/python arcade/etc
  - Sometimes you need to build it
    - Eg python's pygame
- Game loop needs to at least let all game objects update and then draw on screen.
  - All modern libraries/frameworks are 'double buffered' draw to an off screen buffer and then show all at once
- Ebitengine has the game loop in 'RunGame'
  - Which calls Update, Draw and Layout

Update pass 1

- Update gets called in the game loop
  - Which runs *about* 60 times per second
  - So our first update will just move the ship one pixel over
  - Let's try this one out.

```
func (game *firstGame) Update() error {
        game.xloc += 1
        return nil
}
```

Update pass 1

- Update gets called in the game loop
  - Which runs *about* 60 times per second
  - So our first update will just move the ship one pixel over
  - Let's try this one out.
- Hmmm, it looks like we have trouble

```
func (game *firstGame) Update() error {
        game.xloc += 1
        return nil
}
```

Selection in Go

- Selection in Go (AKA if)
  - if <condition/Boolean>{
    - <do this if true>
    - }
- Or
  - if <condition/Boolean>{
    - <do this if true>
    - }else{
    - <do this if false>
    - }
  - No parens around condition, but must have braces {} around body even if one line

Statements

- How does a java statement end?

Statements

- How does a python statement end?
    –

Statements

- How does a python statement end?
  - With the end of the line except for special circumstances

Go Statements

- How Does a go Statement end?
- Reminder:
-
```
package main

import (
     "fmt"
     "io/ioutil"
     "log"
     "net/http"
)
func main() {
     response, err := http.Get("https://news.ycombinator.com/")
     if err != nil{
     log.Fatal(err)
     }
     defer response.Body.Close()
     dataAsBytes, err := ioutil.ReadAll(response.Body)
     if err != nil{
     log.Fatal(err)
     }
     fmt.Print(string(dataAsBytes))
}
```
- Code is a mangling of https://www.devdungeon.com/content/web-scraping-go

Go Statements

- How Does a go Statement end?
- Reminder:
-

```go
package main

import (
	"fmt"
	"io/ioutil"
	"log"
	"net/http"
)
func main() {
	response, err := http.Get("https://news.ycombinator.com/")
	if err != nil{
		log.Fatal(err)
	}
	defer response.Body.Close()
	dataAsBytes, err := ioutil.ReadAll(response.Body)
	if err != nil{
		log.Fatal(err)
	}
	fmt.Print(string(dataAsBytes))
}
```

- More like python
  - End of line except for special circumstances

Selection II

- A more complicated selection example

```
if num := 9; num < 0 {
        fmt.Println(num, "is negative")
    } else if num < 10 {
        fmt.Println(num, "has 1 digit")
    } else {
        fmt.Println(num, "has multiple digits")
    }
```

- Notice two statements in first condition
  - Also variables created in condition are available in all later branches

Back to update

- How could we change update to make sure the image stays on the screen?
  - Let's just loop the image back to the other side rather than reversing it.
  - Let's do that now
  - For example we might

```
func (game *firstGame) Update() error {
	game.xloc += 1
	if game.xloc > 1000{
		game.xloc = 0
	}
	return nil
}
```

  - Possibly clean up to adjust for image size – or leave that for later

Repetition in Go

- In programming theory, two types of repetition, definite and indefinite
  - For and while in most languages
- Go has only for – which it uses for both

Basic Go for loop

- Basic for loop looks a lot like C-like language for loop

```go
func countDown(start int){ //in honor of starship launch
    for counter := start; counter >0; counter--{
        fmt.Println(counter)
    }
    fmt.Println("blastoff")
}
```

- Again
  - no parens around setup, but required braces
  - Scope of variables created in initialization statement only that for-loop

For with only condition

- You can omit the initialization and post part of the for (not the condition)
  - makes it functionally what other languages use while
  - func main() {
    - sum := 1
    - for ; sum < 1000; {
    - sum += sum
    - }
    - fmt.Println(sum)
  - } //From https://tour.golang.org/flowcontrol/2

- Semi colons are optional – can be dropped
- func main() {
  - sum := 1
  - for sum < 1000 {
  - sum += sum
  - }
  - fmt.Println(sum)
- }

The Forever loop

- Since there is no while,
  - Can't have a while True
- Go has something they pronounce "for ever"
- for{
- //Do something forever
- //or at least till we hit a break statement
- }

Let's use a for loop

- Let's add a for loop in draw to draw three ships instead of one.
  - Adjust the hard coded 'magic number' 50 as needed for image

```
func (game *firstGame) Draw(screen *ebiten.Image) {
        screen.Fill(colornames.Blanchedalmond)
        drawOps := ebiten.DrawImageOptions{}
        for i := 0; i < 3; i += 1 {
                drawOps.GeoM.Reset()
                drawOps.GeoM.Translate(float64(game.xloc-50*i),
float64(game.yloc))
                screen.DrawImage(game.player, &drawOps)
        }

}
```

Arrays in Go

- Arrays in Go are interesting
  - Standard fixed size, homogeneous, contiguous data structure
    - Must declare type and size at compile time
      - Eg:
        - `var octoOfInts [8]int;`
        - `var tripleOfStrings [3]string = [3]string{“s”, “t”, “u”}`
  - Array size is part of the type in go
    - And Go is a strongly typed language
    - So what does this mean for parameters in functions?

Arrays in Go

- Arrays in Go are interesting
  - Standard fixed size, homogeneous, contiguous data structure
    - Must declare type and size at compile time
      - Eg:
        - `var octoOfInts [8]int;`
        - `var tripleOfStrings [3]string = [3]string{“s”, “t”, “u”}`
  - Array size is part of the type in go
    - And Go is a strongly typed language
    - So what does this mean for parameters in functions?
    - You need a different function for every size of array if you take an array. These differ:
      - `func reverse(ptr *[8]int){…`
      - `func reverse(ptr *[16]int){...`

Slices

- Arrays are great, but limited, no growth, typing is difficult
- So Go says: ‘use slices’
  - In Go slices are a “view” into a sequence data
    - Usually arrays, but also strings
  - Every slice has an array under it, but slices grow and have variable size.
  - Every slice has:
    - pointer to an array element (first item in slice)
    - len (how many elements in slice
    - cap (how many elements till end of underlying array)

Slices II

- Create an empty slice:
  - `var emptySlice []int`
  - len is 0; emptySlice == nil
- Create a slice with lots of zero values using make
- `names := make([]string, 5, 10)`
  - Makes a sequence of type <first param> with len <second param> and capacity <third param>
    - If cap isn’t specified, len and cap are same
  - Going past len in a slice expands the slice
  - Going past cap, causes *panic*

So let's put in a slice of stuff into our demo

- We will use a slice of coin piles to show this
- Our coinPile struct

```
type coinPile struct {
	pict *ebiten.Image
	xloc int
	yloc int
}
```

- And updated game struct

```
type firstGame struct {
	player   *ebiten.Image
	xloc     int
	yloc     int
	score     int
	treasures []coinPile
}
```

A new function to create coin piles

- I want to put coin piles all over the screen

```
func NewCoins(MaxWidth, MaxHeight int, pict *ebiten.Image)
coinPile {
        return coinPile{
                pict: pict,
                xloc: rand.Intn(MaxWidth),
                yloc: rand.Intn(MaxHeight),
        }
}
```

- Summary:
  - create a struct
  - Use standard library function to randomly assign x,y coordinates inside the screen.
  - No need for random seed if the go version is >= 1.20

Update main

- Lets add these lines to main - ask if you have questions

```
pict, _, err := ebitenutil.NewImageFromFile("coins.png")
if err != nil {
        fmt.Println("Failed to load image", err)
}
allTreasures := make([]coinPile, 0, 15)
for i := 0; i < 10; i += 1 {
        allTreasures = append(allTreasures, NewCoins(1000, 1000, pict))
}
ourGame := firstGame{player: playerPict,
        xloc:      500,
        yloc:      500,
        treasures: allTreasures,
}
```

Removing things from a slice

- To remove an item from a slice
  - Take the first part of the slice up to the item to remove
  - And last part of the slice after the item to remove
  - Use append function to put them together.
  - append takes a slice, and a bunch of stuff to put at the end of the slice
  - … operator right after a slice will unpack a slice into its elements.

If the item to remove is at position i in the slice

```
game.playerShots := append(game.playerShots[:i],
game.playerShots[i+1:]...)
```

For-each loop

- Python (and modern Java) have a 'for each' loop, to iterate over a collection
  - In go use a for range(<collection>) syntax
  - Syntax
    - *for index, item := range collection{*
      - *//do something*
    - *}*

Update draw

- Let's update draw to draw each pile in the slide

```go
func (game *firstGame) Draw(screen *ebiten.Image) {
    screen.Fill(colornames.Blanchedalmond)
    drawOps := ebiten.DrawImageOptions{}
    for i := 0; i < 3; i += 1 {
        drawOps.GeoM.Reset()
        drawOps.GeoM.Translate(float64(game.xloc-50*i), float64(game.yloc))
        screen.DrawImage(game.player, &drawOps)
    }
    for _, pile := range game.treasures {
        drawOps.GeoM.Reset()
        drawOps.GeoM.Translate(float64(pile.xloc), float64(pile.yloc))
        screen.DrawImage(pile.pict, &drawOps)
    }
}
```

The new part

Strings

- We've used strings (in reporting errors if nothing else)
- Strings are officially “an immutable sequence of bytes”
  - Can contain 0 (null byte)
  - Usually interpreted as UTF-8 (unicode)
  - Utf-8 characters are called ‘runes’
  - len(string) returns number of bytes not runes
  - Use utf8.RuneCountInString(<string>) to find out how many characters are in string.

Strings II

- Since strings are immutable
  - How do we build a string with values in it?

Strings II

- Since strings are immutable
  - How do we build a string with values in it?
  - Either with StringBuilder class from standard library
  - Or fmt.Sprintf
    - e.g.
    - Now if g.Score has a value of 3, then scoreString has a value of Score: 3

```
scoreString := fmt.Sprintf("Score: %d", g.Score)
```

Let's look at it all together

- Let's look at our little demo
- All together.
- Ask lots of questions
- Is there anything else you need to know about go?
- We will continue using go and ebitengine to build first protogames, and then games themselves