THE GO PROGRAMMING LANGUAGE

PART 2

LAST WEEK

LAST WEEK

- Background stuff with beards

LAST WEEK

- Background stuff with beards
- Setting up GOPATH

LAST WEEK

- Background stuff with beards
- Setting up GOPATH
- The basics - variables, conditionals, etc...

LAST WEEK

- Background stuff with beards
- Setting up GOPATH
- The basics - variables, conditionals, etc...
- Structs, methods and interfaces

LAST WEEK

- Background stuff with beards
- Setting up GOPATH
- The basics - variables, conditionals, etc...
- Structs, methods and interfaces
- Errors

THIS WEEK

THIS WEEK

- Panic and defer

THIS WEEK

- Panic and defer
- Concurrency

THIS WEEK

- Panic and defer
- Concurrency
- Object composition vs. inheritance

THIS WEEK

- Panic and defer
- Concurrency
- Object composition vs. inheritance
- A game

PANIC

- Sometimes we just want to fail fast - report error, and stop.

```
 1 package main
 2
 3 import "os"
 4
 5 func main() {
 6         _, err := os.Create("/etc/stupidfile")
 7         if err != nil {
 8                 panic(err)
 9         }
10 }
```

DEFER

- Delay function execution until later in the program.
- Useful for cleanup, e.g. closing a file that's been opened.
- Let's look at an example program...

CONCURRENCY

GOROUTINES

- The base components of Go's concurrency.
- Lightweight thread
- To create one, we just call a function with 'go':
- Let's look at an example...

```
1 func printTenTimes(name string) {
2     for i := 0; i < 10; i++ {
3         fmt.Println(name)
4     }
5 }
6
7 func main() {
8     go printTenTimes("Alice")
9 }
```

HOW DO WE COMMUNICATE?

- Answer: Channels!
- These are a bit like ‘pipes’ between goroutines.
- Writing to and reading from channels are blocking.

```
 5 func main() {
 6         messages := make(chan string)
 7
 8         go func() {
 9                 messages <- "Hello!"
10         }()
11
12         msg := <-messages
13         fmt.Println(msg)
14 }
```

BUFFERED CHANNELS

- Normally, we can't send unless there's a waiting receive.
- We can give our channels a buffer size.
- Channels are FIFO.

```
5 func main() {
6         messages := make(chan string)
7         messages <- "Hello!"
8         fmt.Println(<-messages)
9 }
```

(Wrong)

```
 5 func main() {
 6         messages := make(chan string, 2)
 7         messages <- "First message"
 8         messages <- "Second message"
 9         fmt.Println(<-messages)
10         fmt.Println(<-messages)
11 }
```

SYNCHRONISING

- We can use channels to send control signals.
- Remember the first example?

```
1 func printTenTimes(name string) {
2     for i := 0; i < 10; i++ {
3         fmt.Println(name)
4     }
5 }
6
7 func main() {
8     go printTenTimes("Alice")
9 }
```

SYNCHRONISING

- We can use channels to send control signals.
- Remember the first example?

```
1  func printTenTimes(name string) {
2      for i := 0; i < 10; i++ {
3          fmt.Println(name)
4      }
5  }
6
7  func main() {
8      go printTenTimes("Alice")
9  }
```

```
5  func printTenTimes(name string, done chan bool) {
6          for i := 0; i < 10; i++ {
7                  fmt.Println(name)
8          }
9          done <- true
10 }
11
12 func main() {
13         done := make(chan bool)
14         go printTenTimes("Alice", done)
15         <-done
16 }
```

CHANNEL DIRECTIONS

- We can specify the direction of channels we pass to functions - helps with type safety.

```
 5 func printTenTimes(name string, done chan<- bool) {
 6         for i := 0; i < 10; i++ {
 7                 fmt.Println(name)
 8         }
 9         done <- true
10 }
11
12 func main() {
13         done := make(chan bool)
14         go printTenTimes("Alice", done)
15         <-done
16 }
```

SELECT

- Like switch, but for channels.

```
 6 func sleepThenTalk(seconds int, say string, c chan<- string) {
 7         time.Sleep(time.Second * time.Duration(seconds))
 8         c <- say
 9 }
10
11 func main() {
12         c1 := make(chan string)
13         c2 := make(chan string)
14
15         go sleepThenTalk(5, "Five!", c2)
16         go sleepThenTalk(2, "Two!", c1)
17
18         for i := 0; i < 2; i++ {
19                 select {
20                 case msg1 := <-c1:
21                         fmt.Println(msg1)
22                 case msg2 := <-c2:
23                         fmt.Println(msg2)
24                 }
25         }
26 }
```

SELECT

- Select is normally blocking.
- We can make it non-blocking with ‘default’.
- Let’s look at a real-world example...

```
select {
case msg1 := <-c1:
        fmt.Println(msg1)
case msg2 := <-c2:
        fmt.Println(msg2)
default:
        fmt.Println("Waiting...")
}
```

TIMEOUTS

- We can use selects to implement a timeout.

```
14          go sleepThenTalk(50, "Big wait!", c)
15
16  waitloop:
17          for {
18                  select {
19                  case msg := <-c:
20                          fmt.Println(msg)
21                  case <-time.After(time.Second * 3):
22                          fmt.Println("I'm bored...")
23                          break waitloop
24                  }
25          }
26  }
```

RANGE OVER CHANNELS

- We can use range over the contents of a channel.

```
 5 func main() {
 6         c := make(chan string, 3)
 7         c <- "Alice"
 8         c <- "Bob"
 9         c <- "Carlos"
10         close(c) // What happens without this?
11
12         for name := range c {
13                 fmt.Println(name)
14         }
15 }
```

WORKER POOLS

- We can use goroutines and channels to process jobs in parallel, with a pool of workers.

WORKER POOLS

- We can use goroutines and channels to process jobs in parallel, with a pool of workers.
- A worker is a goroutine, and we put the jobs in a channel.

WORKER POOLS

- We can use goroutines and channels to process jobs in parallel, with a pool of workers.
- A worker is a goroutine, and we put the jobs in a channel.
- We define the number of workers available.

WORKER POOLS

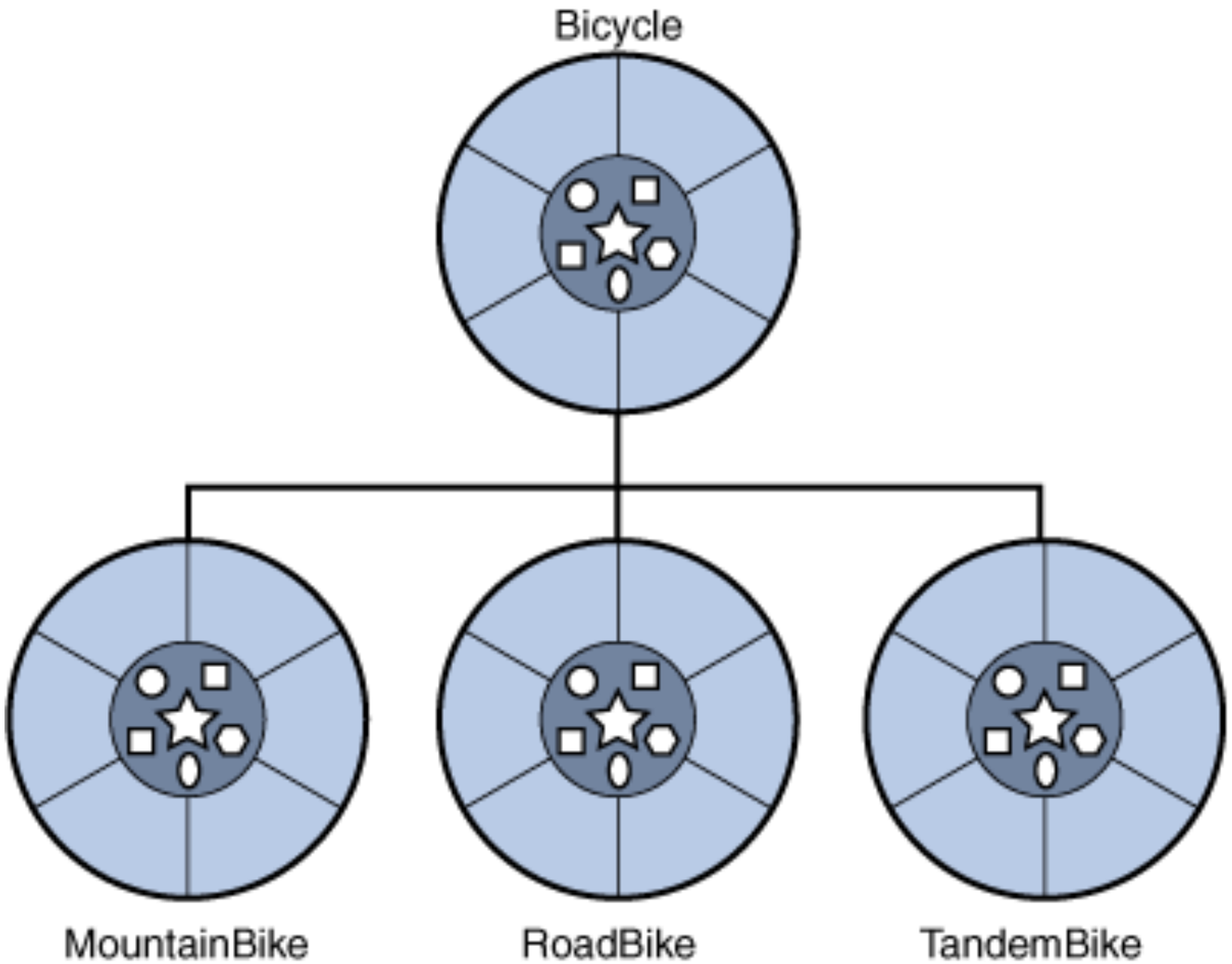
- We can use goroutines and channels to process jobs in parallel, with a pool of workers.
- A worker is a goroutine, and we put the jobs in a channel.
- We define the number of workers available.
- Let’s look at an example…

WORKER POOLS

- We can use goroutines and channels to process jobs in parallel, with a pool of workers.
- A worker is a goroutine, and we put the jobs in a channel.
- We define the number of workers available.
- Let’s look at an example…
- Impressed?

OBJECT COMPOSITION

THE CLASSIC EXAMPLE...



THE CLASSIC EXAMPLE...

```
1  class Bicycle {
2
3      private int wheels;
4      private float speed;
5
6      public Bicycle() {
7          this.wheels = 2;
8          this.speed = 0;
9      }
10
11     public float pedal() {
12         speed += 1;
13         return speed;
14     }
15
16     public float brake() {
17         speed = 0;
18         return speed;
19     }
20 }
```

THE CLASSIC EXAMPLE...

```
1  class Bicycle {
2
3      private int wheels;
4      private float speed;
5
6      public Bicycle() {
7          this.wheels = 2;
8          this.speed = 0;
9      }
10
11     public float pedal() {
12         speed += 1;
13         return speed;
14     }
15
16     public float brake() {
17         speed = 0;
18         return speed;
19     }
20 }
```

```
1  class MountainBike extends Bicycle {
2
3      private int gear;
4
5      public MountainBike() {
6          super();
7          this.gear = 1;
8      }
9
10     public void setGear(int gear) {
11         this.gear = gear;
12     }
13 }
```

THERE IS NO INHERITANCE IN GO.

So what do we use instead?

OBJECT COMPOSITION

- Instead of extending, we keep parents inside children.

```
5  type Shape interface {
6          Area() float64
7          Perimeter() float64
8  }
9
10 type Rectangle struct {
11         width  float64
12         height float64
13 }
14
15 func (r *Rectangle) Area() float64 {
16         return r.width * r.height
17 }
18
19 func (r *Rectangle) Perimeter() float64 {
20         return (2 * r.width) + (2 * r.height)
21 }
```

OBJECT COMPOSITION

- Instead of extending, we keep parents inside children.

```
21 type NamedRectangle struct {
22         rectangle *Rectangle
23         name      string
24 }
25
26 func (r *NamedRectangle) Area() float64 {
27         return r.rectangle.Area()
28 }
29
30 func (r *NamedRectangle) Perimeter() float64 {
31         return r.rectangle.Perimeter()
32 }
33
34 func (r *NamedRectangle) Name() string {
35         return "My name is: " + r.name
36 }
```

OBJECT COMPOSITION

- Instead of extending, we keep parents inside children.

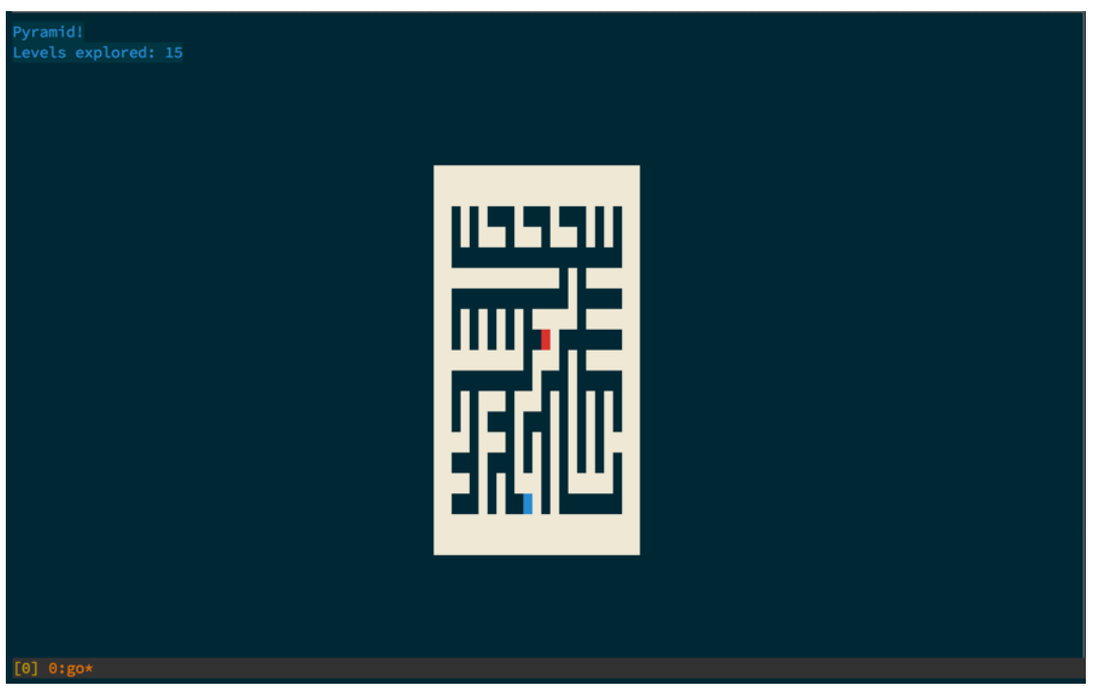
```
40 func main() {
41         r := &NamedRectangle{
42                 name: "Bob",
43                 rectangle: &Rectangle{
44                         width:  10,
45                         height: 20,
46                 },
47         }
48         fmt.Println(r.Name())
49         fmt.Printf("Area: %v\n", r.Area())
50         fmt.Printf("Perimeter: %v\n", r.Perimeter())
51 }
```

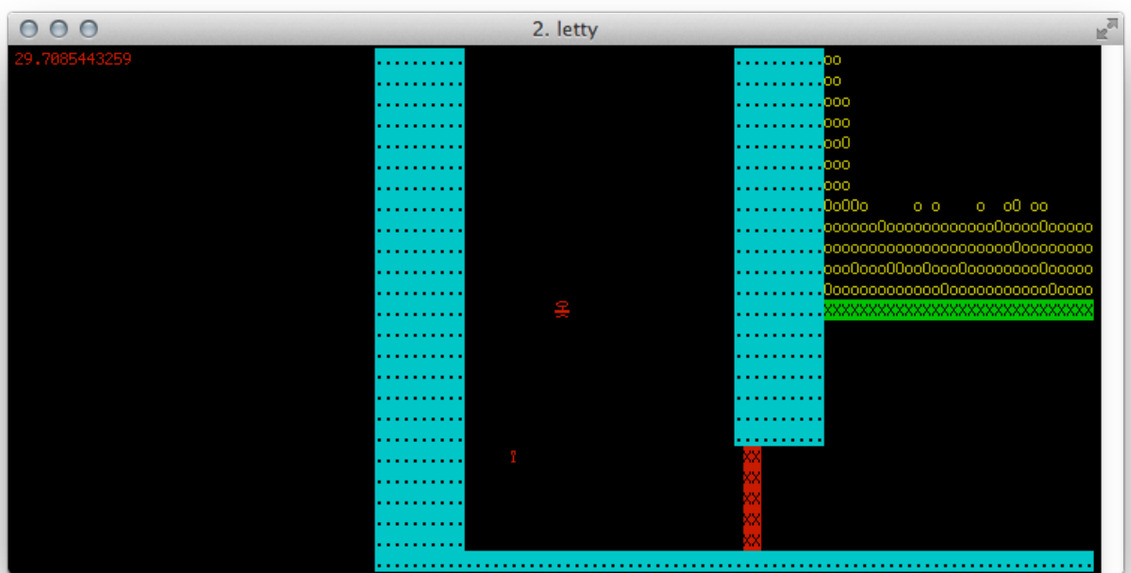
OBJECT COMPOSITION

- Instead of extending, we keep parents inside children.
- Let’s look at this in practice.

TERMLOOP

- A game engine for the terminal.
- Pure Go, uses Termbox.
- Open source: http://github.com/JoelOtter/termloop





THANKS!

@JoelOtter